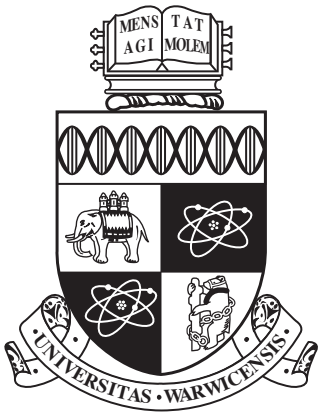


Performance Modelling and Optimisation of Inertial Confinement Fusion Simulation Codes

by

Robert Francis Bird

A thesis submitted to The University of Warwick

in partial fulfillment of the requirements

for admission to the degree of

Doctor of Philosophy

Department of Computer Science

The University of Warwick

February 2016

Abstract

Legacy code performance has failed to keep up with that of modern hardware. Many new hardware features remain under-utilised, with the majority of code bases still unable to make use of accelerated or heterogeneous architectures. Code maintainers now accept that they can no longer rely solely on hardware improvements to drive code performance, and that changes at the software engineering level need to be made.

The principal focus of the work presented in this thesis is an analysis of the changes legacy Inertial Confinement Fusion (ICF) codes need to make in order to efficiently use current and future parallel architectures. We discuss the process of developing a performance model, and demonstrate the ability of such a model to make accurate predictions about code performance for code variants on a range of architectures. We build on the knowledge gained from such a process, and examine how Particle-in-Cell (PIC) codes must change in order to move towards the required levels of portable and future-proof performance needed to leverage the capabilities of modern hardware. As part of this investigation, we present an OpenCL port of the legacy code EPOCH, as well as a fully featured mini-app representing EPOCH. Finally, as a direct consequence of these investigations, we directly apply these performance optimisations to the production version EPOCH, culminating in a speedup of over 2× for the core algorithm.

Acknowledgments

First and foremost, I would like to thank my supervisor, Professor Stephen Jarvis, for all of his help and hard work over the past four years and for allowing me the opportunity to undertake a Ph. D. as well as the opportunities afforded as a consequence.

It is also my pleasure to acknowledge my colleagues, past and present, in the High Performance and Scientific Computing group: Richard Bunt, Adam Chester, Peter Coetzee, James Davis, Simon Hammond, Andrew Mallinson, John Pennycook, Oliver Perks, and Steven Wright. I reserve special thanks for David Beckingsale, there from the start and still there at the end of all things (with hopefully many more to come!). I have also enjoyed the support of many individuals within the Department of Computer Science, including Jane Clarke, Richard Cunningham, Christine Leigh, Roger Packwood, Catherine Pillet, Gill Reeves-Brown, and Paul Williamson. Thank you for all the help and support.

Finally, I would like to thank all those from my personal life who have supported me over the past few years, especially my family: Mom, Dad, Gran, Mike, Laura, Uncle John, Liz, Jasp and William.

Declarations

This thesis is submitted to the University of Warwick in support of the author's application for the degree of Doctor of Philosophy. It has been composed by the author and has not been submitted in any previous application for any degree. The work described in this thesis has been undertaken by the author, unless otherwise stated.

Much of the work presented in this thesis has been published in peer reviewed conferences, workshops and journals. Publication highlights include the following works:

- [8] R. F. Bird, S. A. Wright, D. A. Beckingsale, and S. A. Jarvis. Performance Modelling of Magnetohydrodynamics Codes. In: *Lecture Notes in Computer Science (LNCS), 7587:197-209*, July 2013.
- [9] R. F. Bird, S. J. Pennycook, S. A. Wright, and S. A. Jarvis. Towards a Portable and Future-proof Particle-in-Cell Plasma *Physics Code. In: 1st International Workshop on OpenCL (IWOCL 13)*, May 2013.
- [33] P. Gillies, N. J. Sircombe, R. F. Bird, S. J. Pennycook and S. A. Jarvis. The challenges of porting the Particle-in-cell code EPOCH to new architectures. In: *JOWOG 34, Sandia National Laboratory*, August 2013.
- [34] P. Gillies, N. J. Sircombe, and R. F. Bird. Particle in Cell Simulations at AWE. In: *JOWOG 34, Lawrence Livermore National Laboratory*, April 2014.
- [10] R. F. Bird, P. Gillies, M. R. Bareford, and S. A. Jarvis. Mini-app Driven Optimisation of Inertial Confinement Fusion Codes. In: *IEEE Cluster Workshop on Representative Applications (WRAp 2015)*, September 2015 (Selected for IJHPCA Journal Upgrade).

In addition, research conducted during the period of registration has also led to the following publications which, while not directly focused on Inertial Confinement Fusion simulation, have nevertheless shaped the research presented in this thesis:

- [115] S. A. Wright, S. D. Hammond, S. J. Pennycook, R. F. Bird, J. Herdman, I. Miller, A. Vadgama, A. Bhalerao, and S. A. Jarvis, Parallel File System Analysis Through Application I/O Tracing. The Computer Journal, 56 (2). pp. 141-155. ISSN 0010-4620, 2013

- [94] O. Perks, R. F. Bird, D. A. Beckingsale, and S. A. Jarvis, Exploiting Spatiotemporal Locality for Fast Call Stack Traversal. In: Workshop on High-performance Infrastructure for Scalable Tools, WHIST, Venice, 2012

- [19] E. Carrier, A. Reisner, K. Czuprynski, R. Pavel, P. Grosset, and R. Bird, Evaluating Distributed Runtimes in the Context of Adaptive Mesh Refinement. In: LANL Student Symposium, Los Alamos National Laboratory, United States of America. LA-UR-14-25562, 2014.

- [86] R. Pavel, R. Bird, P. Grosset, K. Czuprynski, A. Reisner, E. Carrier, C. Junghans, B. Bergen, and A. McPherson, Adaptive Mesh Refinement under the Concurrent Collections Programming Model, In: The Sixth Annual Concurrent Collections Workshop (CnC-2014), 2014.

Sponsorship and Grants

The research presented in this thesis was made possible by the support of the following benefactors and sources:

- The EPSRC grant "A Radiation Hydrodynamic ALE Code for Laser Fusion Energy" (EP/I029117/1)
- The UK Royal Society through their Industry Fellowship Scheme (IF090020/AM).
- The AWE/Warwick Centre for Computational Plasma Physics

Additionally, I would like to thank all of the people who made the experimental aspects of this research possible, including but not limited to: those at EPCC who maintain ARCHER; those at the Open Compute Facility at Lawrence Livermore National Laboratory which provides access to Sierra; and those at the University of Warwick who facilitated access to Minerva.

Abbreviations

ALE	Arbitrary Lagrangian-Eulerian
AoS	Array-of-Structs
AoSoA	Array-of-Structs-of-Arrays
ATAM	Architecture Tradeoff Analysis Method
AVX	Advanced Vector Extensions
CFL	Courant-Friedrichs-Lewy
CnC	Concurrent Collections
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
FDTD	Finite-Difference Time-Domain
FLOP/s	Floating-Point Operations per Second
FMA	Fused Multiply-Add
GB/s	Gigabytes per Second
GFLOP/s	Giga-Floating-Point Operations per Second
GHz	Gigahertz
GPU	Graphics Processing Unit
HPC	High Performance Computing
ICF	Inertial Confinement Fusion
ILP	Instruction Level Parallelism

LLNL Lawrence Livermore National Laboratory

LPI Laser-Plasma Interaction

MHD Magnetohydrodynamics

MIMD Multiple Instruction, Multiple Data

MISD Multiple Instruction, Single Data

MPI Message Passing Interface

NIF National Ignition Facility

NUMA Non-Uniform Memory Access

OpenCL Open Computing Language

PIC Particle-in-Cell

PFLOP/s Peta-Floating-Point Operations per Second

PPC Particle-Per-Cell

QED Quantum Electrodynamics

SDK Software Development Kit

SIMD Single Instruction, Multiple Data

SIMT Single Instruction, Multiple Thread

SISD Single Instruction, Single Data

SMT Simultaneous Multi-Threading

SoA Struct-of-Arrays

SPMD Single Program, Multiple Data

SSE Streaming SIMD extensions

SST Structural Simulation Toolkit

TDP Thermal Design Power

VLIW Very Long Instruction Word

WARPP Warwick Performance Prediction

Contents

List of Figures

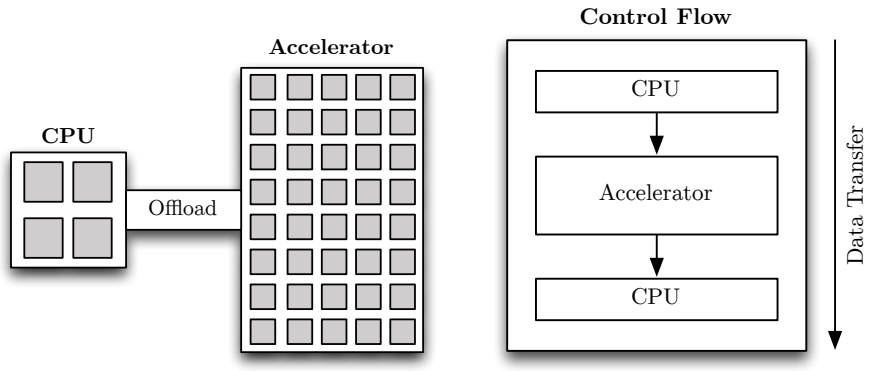
List of Tables

CHAPTER 1

Introduction

Computational experimentation is a vital tool of modern scientific investigation. It allows for full-resolution reproduction of real world phenomena, including events which cannot be investigated closely enough through observation alone. The computational simulation of such events offers a safe and practical method to perform initial scientific investigation, even into areas in which experimentation could be extremely dangerous or prohibitively expensive. Such simulations are often so complex in nature that they cannot be completed on desktop computers alone, and researchers have to rely on the computational power offered by supercomputing facilities. *Supercomputers* are large machines, maintained specifically to facilitate such complex calculations and are typically many orders of magnitude more powerful than desktop computers. An entire research field has been formed around the efficient use and development of powerful supercomputers and the algorithms to support them; today, this field is commonly referred to as High Performance Computing (HPC).

Supercomputing was first realised in the early 1960s, and has continued to develop ever since. The delivery of machines such as the Atlas Computer in 1962, and the CDC 6600 in 1964 were amongst the first landmark events in supercomputing's long history [65, 108]. Despite the lack of similarities between such machines and current supercomputers, they undeniably played a strong role in the development of computational simulation, and shaped what we know today as HPC. Modern supercomputers are able to perform around 10 orders of magnitude (10^{10}) more Floating-Point Operations per Second (FLOP/s) than their historic counterparts, and have increasingly been able to incorporate off-the-shelf commodity components in their design [109]. Current supercomputers



(a) Hardware (b) Data Movement

Figure 1.1: Accelerated Supercomputing architecture from the perspective of (a) Hardware; and (b) Data Movement.

typically feature vast numbers of compute nodes, interconnected with a high-speed network specifically designed for inter-process communication. Historically, such compute nodes have typically featured traditional Central Processing Units (CPUs), with any inter-process communication handled explicitly by the programmer via message-passing. This, however, will likely change with the increased adoption of heterogeneous computing. Hardware manufacturers have started to pair traditional CPU architectures, with specialised *accelerator* (or *co-processor*) architectures to which the CPU can offload computation. These accelerators can either be in the form of commodity Graphics Processing Units (GPUs), or more specialised hardware such as the Intel Xeon Phi product range. Figure 1.1 highlights this paradigm, with work being offloaded (typically over a PCIe bus) to the accelerator from the host CPU.

As of June 2015, 90 of the TOP500 supercomputers utilised heterogeneous accelerators to increase their computational performance – including four of the top ten [110]. Figure 1.2 provides a summary of the increase in accelerator use amongst the worlds fastest supercomputers over the last ten years. Since 2006, the use of heterogeneous accelerators has increased steadily; a trend which is predicted to continue as more supercomputing centers seek the increased

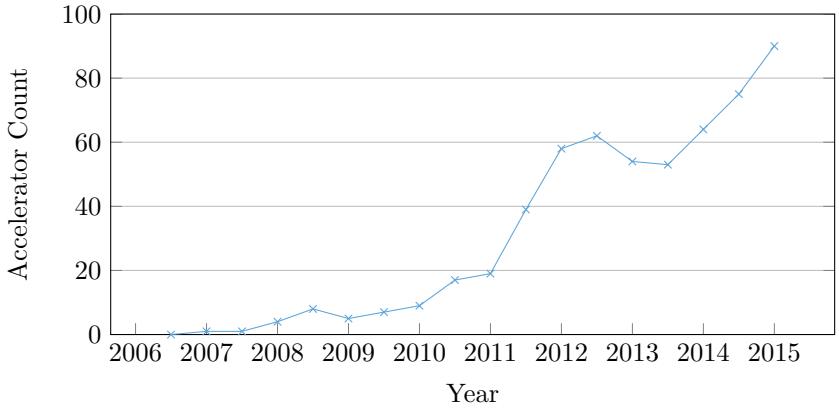


Figure 1.2: Accelerator count in the TOP500 Rankings from 2006 to 2015[1].

performance and power efficiency offered by heterogeneous computing [58].

The demand for more accurate and complex simulation is one of the major driving forces behind the evolution of supercomputers. Given sufficient compute resources, current scientific simulations are often accurate enough to simulate physical reactions at the atomic level, and can include direct particle interaction.

As the need for more detailed simulations continues, hardware designers are being faced with the increasingly difficult task of delivering supercomputer performance which can match this. As we move into the so-called 'exascale era' [27], compute hardware is more complex than it has ever been and the hardware improvements delivered by increasing transistor counts are less impactful than ever before. This forces developers to invest more heavily in software-level improvements to achieve increased code performance. Application engineers strive for maximum performance through code optimisation and algorithmic research, but achieving peak performance is made increasingly more difficult by the diverse range of hardware available in modern HPC platforms and by the large scale of the codes which need to be changed in order to leverage it.

This thesis presents an investigation into developing an understanding of how to overcome these issues in the context of Inertial Confinement Fusion (ICF) simulations. It aims to offer techniques to enable such codes to run well on both

[1]The decrease in accelerator count in Figure 1.2 can be accounted for by the aging and decommission of systems featuring the popular NVIDIA Tesla 2090 GPU

current and future generations of hardware, ensuring that as supercomputers move forward, so too do the codes which they need to support.

1.1 Motivations

Recent decades have seen vast improvements to compute hardware; improvements which have historically been very challenging for code developers to fully utilise. Large bodies of work have been formed over decades focusing solely on performance optimisation [9, 10, 16, 102]. Such optimisation procedures are often highly complex, requiring extensive domain knowledge and an in-depth understanding of the code. As CPU complexity grows, so too does the gap between current code performance and that which is attainable. Often, simulations which use such CPUs were developed in such a way that they were, perhaps unintentionally, closely related to the hardware available at their inception. In order to exploit the increase in CPU performance and functionality afforded by modern CPU design, it is first essential to understand existing code performance; a task most typically done through profiling, benchmarking, and modelling.

The work in this thesis focuses on the major challenges facing legacy ICF codes to deliver performance on both modern and future hardware. Specifically this includes the changes required to multiple legacy plasma physic simulation codes, including those in the areas of Particle-in-Cell (PIC) simulation and Magnetohydrodynamics (MHD) simulation. ICF is an important area of computational research, driving real-world scientific discovery and offering the possibility of solving the world's energy crisis. ICF experiments are typically high-power in nature, with select sites being able to deliver petawatt focused beams with approximately 10,000 times more power than the UK National Grid (during pico-second pulses). Any effort to better understand and improve the performance of such simulations not only stands to save computational time and energy, but also reduces time to scientific solution and discovery. Such

optimisations are key to this thesis, in which the ability of performance models to provide important insights into code performance is demonstrated. The development of a mini-app is presented, which represents their ability to guide code optimisation and to be a powerful research tool. Throughout, the fundamental problems with classical techniques are highlighted, motivating the need for either a change in the core algorithm or an alternative way of expressing parallelism to be developed.

1.2 Thesis Contributions

This thesis makes the following specific contributions to ICF code performance engineering and optimisation:

- A novel performance model for the MHD code Lare is presented. It is the first known predictive performance model for Lare, and allows for the prediction of runtime across a range of current and future architectures based on minimal parameter inputs. It is validated on two HPC systems with an accuracy of greater than 90% for both weak and strong scaled problems on over three thousand cores. This model is then extended to investigate the effects of modifying the Lare code-base to include aspects of Arbitrary Lagrangian-Eulerian (ALE) methods, a change which would further extend the range and depth of the physical phenomena simulated by Lare, to better facilitate ICF research. The techniques used for this modelling processes are applicable to other ICF codes, including those codes discussed later in this thesis, such as the PIC code EPOCH;

- The development of the first documented port of the EPOCH code-base to accelerator architectures is shown. An Open Computing Language (OpenCL) mini-benchmark which represents the key compute kernel of the PIC algorithm used in EPOCH is developed. The use of accelerator specific optimisations to EPOCH is explored, including the efficient use of Single Instruction, Multiple Thread (SIMT) computation and explicit

shared memory. A benchmarking comparison is then performed across a variety of hardware, including a range of CPUs, and three generations of NVIDIA GPUs (Tesla, Fermi and Kepler). This work culminates in a discussion about the code changes needed for EPOCH and other ICF codes to successfully utilise accelerated heterogeneous hardware;

- A fully featured mini-application which represents the PIC code EPOCH is developed. It is the first publicly recorded mini-app explicitly targeting a Finite-Difference Time-Domain (FDTD) PIC plasma physics code. The mini-app is then validated and verified as being able to fully recreate a range of physical simulations available within EPOCH, with a particular focus on Laser-Plasma Interaction (LPI). The mini-app is then used to investigate known performance issues within EPOCH, including poor time-step scaling of long runs, and high levels of cache misses due to particle store fragmentation;

- Finally, as a direct impact of the knowledge gained through all previous work, the previously discovered optimisations for PIC codes are mapped back to the legacy EPOCH code base. This allows EPOCH to scale linearly in a time-step basis, exhibit improved particle per cell scaling, and demonstrate vastly improved cache hit rates and memory locality. Overall this culminates in a speedup of over 2× for the production EPOCH algorithm, an improvement which will decrease the time to solution for novel scientific discovery.

1.3 Thesis Overview

Chapter 2 presents an account of the basic underlying computational theory that underpins all of the work contained within this thesis. It details the governing laws and equations of parallel computing, provides a background to benchmarking and profiling, and introduces the concepts required for performance modelling.

Chapter 3 discusses the theoretical and practical aspects of parallel computing. It highlights the different types of parallelism available in modern hardware, and discusses the mapping between theory and implementation, as well as outlining some of the challenges that arise when trying to fully exploit the available parallelism offered by modern hardware. Much of the work in this thesis focuses on overcoming such challenges which arise as a direct consequence of the complex interactions between levels of parallelism.

Chapter 4 presents a background to ICF codes and the typical structure of such scientific simulations. It describes the operation of two codes: EPOCH, a PIC plasma physics code; and Lare, an MHD code – both of which represent key components of typical ICF research workflows. Further, Chapter 4 highlights key areas of the codes which dominate their performance profile; and provides intuition into how the codes operate.

Chapter 5 presents a performance model of the MHD code, Lare. The model can accurately predict runtimes for both current and future generations of compute hardware. It details the processes of producing such a model, and validates it on two HPC systems. The model is then used to make performance predictions about an optimised version of Lare which includes aspects of ALE methods. The processes presented can be applied to all similar codes, with the most applicable being other codes relating to ICF research.

Chapter 6 documents the initial investigation into the use of accelerators to enhance PIC simulations. Specifically, it presents a case study of porting EPOCH to OpenCL in the form of a mini-benchmark, and offers conclusions about how such a process applies to other plasma physics codes. It highlights key problems in the legacy PIC algorithm which limit its suitability for accelerator hardware, and offers solutions and techniques to overcome these.

Chapter 7 presents the development of the first known FDTD PIC mini-app, known as miniEPOCH. It builds on the understanding gained in Chapter 6 to show how such an investigation can culminate in improved code performance of the parent application. The mini-app is used to investigate a variety of novel optimisations, at each step showing how they map back to the original code, ultimately culminating in improved application performance.

Chapter 8 concludes this thesis with a discussion of the implications of the work presented. The limitations of the research presented are outlined, with final remarks about possible opportunities for future work being outlined, as well as the inclusion of some preliminary results.

CHAPTER 2

Performance Analysis and Modelling

Increased computational performance has always been at the forefront of scientific development [72]. For centuries, theorists have sought algorithmic improvements to reduce computational complexity, with the terms algorithm and algebra having been established over 1200 years ago. The pursuit of increased computational performance is most commonly motivated by the desire for increasingly complex and accurate scientific simulation. As the prominence of High Performance Computing (HPC) and scientific computing increases, so too does the importance of developing new algorithms and methodologies for achieving efficient computation. In this Chapter an overview of the current state of HPC is provided, as well as introducing the fundamental concepts that underpin both the work in this thesis and computational simulation as a whole.

2.1 The Laws of Parallel Computing

While parallel computing is often perceived to be a very practical domain, it is underpinned by a rich background of theory. The following equations govern the behaviour and limits of all computation, and most importantly can be applied to give an accurate insight to application performance.

2.1.1 Speedup

Speedup offers a measure of scalability for an application; it demonstrates how well an application makes use of increasing numbers of processing elements through a comparison of runtimes. Conceptually it is the ratio of serial runtime

(T_s) to parallel runtime (T_p). This can formally be expressed as:

$$S_p = \frac{T_s}{T_p} \tag{2.1}$$

This is a useful metric, most commonly used to quickly assess how a code will scale with increasing compute resource and has gained popularity because of the ease in collecting the required numbers. A code which exhibits a near linear speedup is said to scale well.

2.1.2 Parallel Efficiency

Parallel efficiency leverages the concept of speedup, and uses it to measure the proportion of the available parallel improvement a code is able to obtain when running on N parallel processors.

$$E_p = \frac{S_p}{N} \tag{2.2}$$

Typically the measure of parallel efficiency has a value between 0 and 1, with 1 representing ideal parallel efficiency. Any number smaller than one shows a loss in parallel efficiency. It is important to note, however, a value of greater than one is possible to obtain, and is referred to as a super-linear speedup. Such speedups are not typical for complex algorithms, and are often a side effect of a change exploiting machine-specific architecture, causing a discrepancy with the theoretical improvement.

2.1.3 Amdahl's Law

In 1967, Gene Amdahl proposed *Amdahl's law* [2]. An equation which governs how parallelism can affect overall code performance and describes the maximum speedup achievable through increasing parallelism. The theory states that for a parallel code with serial fraction F_s and parallel fraction F_p (equivalent to $1 - F_s$), the speedup for n processors will always be dominated by F_s for

increasingly large values of n. This can be more formally expressed as:

$$S_N \leqslant \frac{1}{F_s + \frac{F_p}{N}} \tag{2.3}$$

For many years the HPC industry has benefited from increasing numbers of processors (N), whilst the time to complete the serial code portion (F_s) has simultaneously benefited from increasing single core performance and more complex Central Processing Units (CPUs). The trend of increasing single core performance has diminished, meaning that further improvements to serial code runtime must come from algorithmic changes. Parallels are often drawn between Amdahl's law and the practice of *strong scaling*, during which additional processors are added in an attempt to decrease the time to solution for a given scientific simulation.

2.1.4 Gustafson's Law

Gustafson's law [39] states that despite the bounds provided by Amdahl's law, increasingly large data sets can be efficiently parallelised, as the parallel work fraction increases. This can be most easily conceptualised by considering a fixed time window, and observing how the amount of work solvable in that time changes with the increased availability of parallel hardware. As more processors (N) are made available, larger problem sizes can be solved within the fixed time window despite the presence of a serial code portion (F_s). This can formally be expressed as:

$$S_N = N - F_s \times (N - 1) \tag{2.4}$$

Gustafson's law is often compared to the practice of *weak scaling*, where more processors are added to the system to solve a proportionally larger problem in the same time frame. This provides a direct counterpoint to Amdahl's law and the strong scaling it represents.

2.2 Benchmarking

The performance of compute hardware has historically been measured in Floating-Point Operations per Second (FLOP/s); a metric which is intended to represent the bottleneck in scientific computation. The theoretical peak FLOP/s figures quoted by hardware manufacturers provide an upper bound on application performance. These figures, however, are not representative of how well real-world applications can exploit the power of compute hardware; as many applications only achieve a fraction of this quoted rate.

In order to achieve maximum FLOP/s performance, all functional units of the CPU must be consistently used. Typically this means that at any point in time, the CPU must make full use of the Single Instruction, Multiple Data (SIMD) width, whilst simultaneously dispatching a Fused Multiply-Add (FMA) every clock cycle, per core. Any operation which deviates from this represents a reduction in performance, including any stalls caused by operations such as reading from memory. It is widely understood that although theoretical FLOP/s rates serve as a good single metric indicating potential performance, it is a poor measure of real-world application performance [25, 36, 109]. The process commonly known as *benchmarking* seeks to address this issue.

Benchmarks are specialised applications designed to measure some aspect or property of a computational system. Benchmarks are often designed to collect performance data which is representative of real world applications, and to provide metrics which can then be compared and used to evaluate the suitability of the hardware. A variety of benchmarks exist to try and categories the absolute performance of a supercomputer, each targeting very specific hardware features. These results can then be collated and used to rank supercomputers, as seen in schemes such as the TOP500 [109] and the Graph500 [36]. Such benchmarks include: LINPACK [26], a linear algebra benchmark used to rank supercomputer performance in the TOP500 list; STREAM [71], a memory benchmark used to evaluate sustained memory bandwidth throughput; and SkaMPI [96], a network

benchmark used to evaluate the performance of the Message Passing Interface (MPI) library and the machine interconnect.

Despite the availability of these component-specific benchmarks, it is difficult to combine the metrics in order to make conclusions about more complex algorithms which rely on their combined behaviour. One possible solution is the development of domain or application specific benchmarks, which exhibit analogous computational behaviour to production applications. These benchmarks can be more light weight and portable, and even represent commercially sensitive code. It is common to see benchmark suites which represent the most prominent areas of scientific investigation, examples include the NAS parallel benchmark suite [6]; the Rodinia benchmark suite [20]; the SPEC benchmark suite [111]; and the NESRC benchmark suite [3, 77]. Such benchmarks suites can play an important role in procurement procedures, as they can capture a set of metrics which represent the workload of the relevant supercomputing centre. The NERSC benchmarks are designed specifically for this purpose, whilst many of the other suites are intended for more general use in the classification of machine performance.

2.3 Profiling

Whilst benchmarking provides powerful insights into how codes may perform on different machine configurations, their use only goes part way into explaining code performance for specific applications. Benchmarks often fail to capture sufficiently low-level detail to identify the root cause of application improvement, instead only showing how runtimes vary. *Profilers* are tools which address this issue by monitoring an application as it runs, and collecting a range of performance metrics during its execution. These metrics can then be analysed, to gain an insight and understanding into the program's behaviour. Common metrics include multi-level execution time tracking [35], memory behaviour analysis [78, 93], network communication pattern identification [47], and I/O oper-

ation tracing [18, 115].

It is important to understand code performance for a given application, and profilers are a powerful way to achieve this. Throughout this thesis, the investigations rely heavily on the profiling of codes, and predominantly on the hand instrumentation of source code after an initial investigation has been carried out using traditional profiling tools such as those previously described. The information gained from these profiling efforts provides great insight into code performance, and allows future endeavors to be better directed. The concept of performance modelling discussed in Section 2.5, and used throughout Chapter 5, relies heavily on the concept of profiling.

2.4 Representative Applications

Representative applications are a set of small, self-contained, programs which directly correspond to a parent application, whilst having reduced complexity. They act as highly specialised benchmarking tools, targeted specifically at a sub-component of the parent application. Broadly, representative applications can be divided into two major subgroups: mini-applications (mini-apps); and proxy applications (proxy-apps). Mini-apps represent their parent application by performing the same algorithmic computation as the parent application (perhaps with different input), whilst proxy-apps represent their parent application by performing analogous computation that is expected to replicate the same stress the parent application would put on the machine. There are many factors which may motivate the development of a representative application, the most prominent being benchmarking, optimisation, and architecture evaluation. Additionally, proxy-apps can represent closed source or classified codes without giving away any intellectual property.

Figure 2.1 shows how representative applications relate to both production applications and traditional benchmarks (micro-benchmarks). The closer a benchmark is to the original production application, the more representative

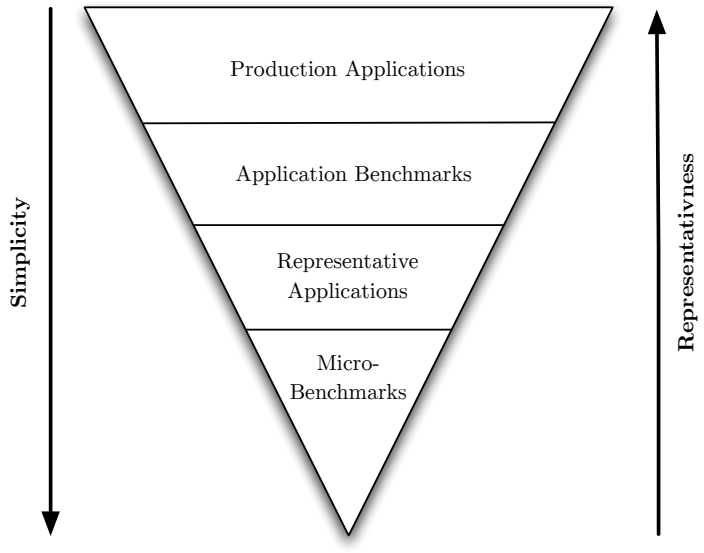


Figure 2.1: Representativeness and Simplicity of Application Scale.

it is, but this comes at the cost of increased complexity. Mini-apps are important as they occupy a good balance of representativeness, without being overly complex. They allow ideas to be tested in a more manageable environment than a production code, but still allow the findings to be sufficiently meaningful that they can then be transferred back into the parent application. Given their significantly smaller size, mini-apps are ideally suited for testing new techniques or programming models, which would otherwise be intractable if attempted in the parent code.

Despite the benefits of mini-apps being identified as early as 1991 [6], the re-emergence and use of mini-apps has gained greater traction in recent years [44]. There are now examples of mini-app use in co-design, code optimisation and porting, and in the exploration of new programming languages and paradigms.

The use of mini-apps for code optimisation is exemplified in the work by Karlin et al., in which the authors use the mini-app LULESH (Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics) to demonstrate the optimisation of multi-material hydrodynamics simulations. By using their mini-app

to better focus optimisation efforts, they increase LULESH's vector instruction utilisation by a factor of 8, reduce the number of memory reads by 62%, and reduce the overall application memory footprint by 19%. These improvements were then mapped back to ALE3D to achieve a 20% reduction in overall application runtime [53, 54, 66, 80], performance gains that had remained undetected until the mini-app investigation.

Further studies demonstrate the use of mini-apps to rapidly investigate both hardware platforms and programming paradigms. Lavallée et al. demonstrate how the mini-app HYDRO was used to investigate multiple hardware platforms available through the PRACE Tier-0 Research Infrastructure [64]; similarly, the size of the mini-app greatly aided their development of several code variants of HYDRO, including those employing MPI, OpenMP, CUDA, OpenCL, HMPP and UPC – a task that would have been infeasible using the full production application. The availability of such a diverse range of code implementations allowed the authors to evaluate a variety of heterogeneous hardware and assess its viability as a future platform for the parent application RAMSES [107], an EU-funded computational astrophysics package used for the study of large-scale structure and galaxy formation. Such studies demonstrate the importance of mini-apps as a tool to enable studies in code portability, scaling and performance.

Mini-apps are increasingly being developed within open source frameworks, thus allowing the HPC community to benefit from their development. Projects such as Mantevo [7, 44] and the UK Mini-App Consortium (UKMAC) [92] exist to provide centralised repositories where collections of mini-applications can exist, selected to represent key scientific areas supported by HPC simulation.

2.5 Performance Modelling

Performance modelling refers to a set of techniques which allow computer scientists to reason about code performance through the invocation of a model.

Typically this involves the development of a conceptual model to represent the code, with the aim of offering highly accurate runtime predictions. Such models can only be developed once a good understanding of the code is formed, a task which relies heavily on the previously outlined techniques of profiling and benchmarking.

Once a model has been established, it can be used to make predictions about how algorithmic changes might affect the performance of an application, and also to predict how architectural differences may impact code runtime. The use of modelling has been demonstrated to identify performance bottlenecks [46], evaluate algorithmic changes [8], predict application performance when ported to new architectures [24], and to quantify the effect of improved communication behaviour on code performance [45]. The concept of performance modelling is key to this thesis, and is discussed further in Chapter 5. Broadly, the techniques for developing a performance model used in this thesis can be split into two categories: (i) *analytical models* which capture application performance as a set of equations; and (ii) *simulation*, where a code model is run on simulated hardware. A third category of performance models exists, which capture mathematical modelling languages such as Petri nets [76], and Markov chains [12].

2.5.1 Analytical Model

The general runtime of a parallel application can be described by Equation 2.5, which states that the total runtime is a combination of the compute time, communication time, synchronisation time, and any time lost during overhead operations. Each term can be constructed from a series of observations and sub-models which capture the aggregate contributions of each application subcomponent which, when combined, represent the overall application runtime.

$$\begin{aligned} T_{total} = (T_{compute} + T_{comms} - T_{overlap}) \\ + T_{synchronisation} + T_{overhead} \end{aligned} \tag{2.5}$$

Compute costs are typically obtained during empirical investigation using benchmarks, along with custom timers. Message costs are typically captured by a communication model parameterised for a given network, combined with message counts and sizes. Equation 2.5 is often simplified to assume negligible communication overlap and application overhead, as described in Equation 2.6.

$$T_{total} = T_{compute} + T_{comms} \tag{2.6}$$

When developing a performance model it is usual to start with the simplistic case of a serial run, as it contains no communications. In doing this, Equation 2.6 can be further simplified to only include terms relating to the cost of computation. This can then be expanded as shown in Equation 2.7 to present the per work unit function costs, known as *grind times* (w_g).

$$T_{compute} = \sum w_g \tag{2.7}$$

To obtain these grind times, timing data must be gathered. This can either be done using a profiler such as gprof [35] or scalasca [31], or alternatively the code can be instrumented manually. Once these grind times have been calculated, they can be put back into Equation 2.7 to obtain the total compute time. A per platform communication model must then be generated based on empirical machine latency and bandwidth, for varying message sizes to account for protocol changes. The communication and computation terms can then be combined to satisfy Equation 2.6. Finally, where possible, any additional synchronisation and overhead costs must be established. In practice however, these costs are often represented in the compute and communication costs. The purely mathematical nature of analytical models brings both clear advantages and disadvantages. Their simplicity means that they can be evaluated near instantaneously, where new configurations can be re-evaluated by substituting different parameters into the model equations. This, however, comes at the cost of reduced complexity and accuracy for complex simulations.

A great deal of literature exists showcasing the success of performance modelling efforts for assessing both current and future architectures [1, 23]. Hammond et al. show how performance modelling can be used to provide a comparison between two different systems, and use this comparison to aid procurement decisions [42]. They show that the ability to make predictions at scale can be more valuable than the information obtained from small scale benchmarks. Herdman et al. use a performance model of an industry strength hydrodynamics benchmark to provide guidance for the procurement of future systems [43]. The authors use their performance model to generate a range of predicted values for comparison, spanning multiple architectures and compiler configurations. In addition to allowing the assessment of current architectures, performance modelling also plays a vital role in enabling us to look at the performance of applications on future architectures at scale. Pennycook et al. show how performance modelling can be used to provide an insight into how applications will perform on a variety on architectures, highlighting the potential benefits of using many-core architectures [90]. Finally, Mudalige et al. show that performance modelling can be applied to emerging distributed memory heterogeneous systems to provide an analysis of the performance characteristics and to accurately predict runtimes for an application [32, 75].

2.5.2 Simulation

Simulators aim to further improve on the prediction accuracy of analytical models by simulating hardware interactions to capture additional system-behaviour details. The concept of simulation covers a broad range of possible scenarios, from a full featured hardware simulation, to combining simulating network events with an analytical compute model. The most common form of performance model simulation is a 'macro' level simulation, in which computation times are collected empirically and communication costs are derived from a fully simulated topology primed with an empirical baseline. This approach tends to give compute results with comparable accuracy to that of an analytical model,

but offers improved communication accuracy. If desired, the network infrastructure used in the simulation can be changed to investigate how different networks may affect the runtime, perhaps even simulating theoretical network topologies and contention. Such improvements do not come without a cost, as simulations tend to take significantly more time to return their predictions as they run through the control flow code of the simulation. By having this closeness between software and hardware it allows for greater performance optimisation of both, as seen in the co-design approach that is being used to move towards exascale [41, 48, 98].

The simulation work presented in this thesis makes use of Structural Simulation Toolkit (SST)/macro [48]. This is just an example of one such toolkit, with others such as the Warwick Performance Prediction (WARPP) [40, 41] toolkit offering similar functionality. SST was chosen because it is well established in the field, and has been shown to offer a range of accurate and useful features. In addition to this a tool to generate un-primed SST/macro models from Fortran source code was also developed in order to facilitate the development of multiple models.

2.5.3 Modelling Languages and Queuing Theory

The aforementioned techniques only represent a small portion of those available in the HPC literature. Other common techniques include the use of Petri nets [76] and Markov chains [76]. A Petri net is a bipartite graph in which the nodes represent transitions or places. Directed arcs connect these nodes, describing pre and post conditions for the transitions. Distributed systems can be modeled as transitions and places, connected with weighted arcs. Their operation is not dissimilar to that of Markov chains, in which a statistical model is built from interconnected state spaces to represent a distributed system. A Markov chain, however, is a stochastic process which is 'memory-less'.

2.6 Summary

This Chapter presents a description of the governing equations that have provided a theoretical basis for evaluating application performance for over 50 years. These equations can be combined with the techniques of profiling and benchmarking in order to aid in real world performance evaluation and code understanding. These techniques can also be combined to develop powerful tools for understanding code performance, such as performance models.

In this thesis we use the described techniques as the foundation for our optimisation efforts and show the importance of understanding code performance through the development of a performance model. This high level of code understanding is crucial for Inertial Confinement Fusion (ICF) simulations to make effective use of modern and heterogeneous hardware.

CHAPTER 3

Parallel Hardware and Programming Models

Modern hardware is highly parallel, and seeks to employ parallelism at every level of its design. This can range from very low-level parallelism, such as Instruction Level Parallelism (ILP), to higher levels of parallelism, such as task-based threaded execution. In order for good code performance to be achieved, each level of parallelism must be utilised to its maximum potential. Not only does this require high levels of understanding, it often represents great effort required by code specialists. In this Chapter we discuss modern hardware features and programming models, and describe how these practical areas correspond to the theory. We also detail the benchmarking platforms used to gather the data presented in this thesis.

3.1 Flynn's Taxonomy

In 1966 Michael Flynn introduced a classification for computer architectures with the goal of allowing a distinction to be made between different theoretical models for parallel hardware, and the ways in which they consume data [30]. The taxonomy is centered around the idea that under different schemes of operation, it is possible for both data and instructions to be executed in parallel. By considering these as either serial (single) or parallel (multiple), a four-way classification arises commonly known as *Flynn's taxonomy*. Figure 3.1 shows Flynn's taxonomy, and highlights the possible combinations of single and parallel execution of both data and instructions. In the context of Flynn's taxonomy, an instruction can be thought of as a basic data operation such as an add or a multiply, and data can be visualized as one or more floating point numbers to which an instruction can be applied.

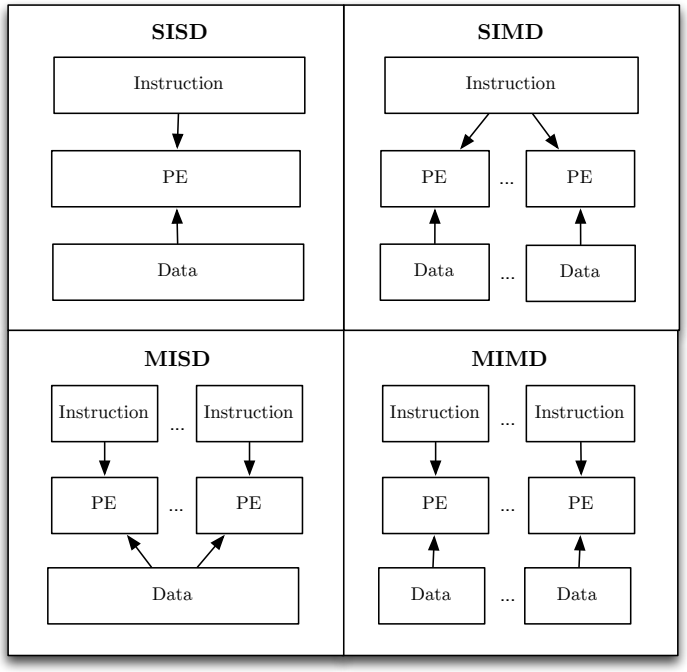


Figure 3.1: Flyyn's Taxonomy, showing the application of parallel processing elements (PEs) to instructions and data.

Single Instruction, Single Data (SISD) represents the most basic form of computation, in which at most a single instruction operates on a single piece of data. This is typical of the very first computers that emulated the behaviour of a person performing arithmetic. It was not until the demand for increased computational speed outstripped the performance offered by SISD processors that parallelism became commonplace.

Single Instruction, Multiple Data (SIMD) describes a mode of parallel operation whereby a single operation, such as an addition, can be applied to multiple values or items of data at once. This can most commonly be seen in vector calculus, which represents many of the core algorithmic operations behind the calculations required by computational simulations. An example im-

plementation of SIMD computation in current hardware is the Advanced Vector Extensions (AVX) instruction set. It includes the support for instructions such as `VADDPD`, in which a single instruction (add) is applied to multiple data points simultaneously to achieve n-way data parallelism. Under the SISD paradigm, these instructions would be processed sequentially, one data element at a time.

Multiple Instruction, Single Data (MISD) is the least frequently seen type of parallel computer architecture. Whilst it is theoretically possible to perform multiple instructions on a single data item, it is realistically challenging for hardware to manipulate a single data value in parallel. These difficulties mean MISD computers are used to fulfill special use cases only, and are not typically found within the scope of High Performance Computing (HPC).

Multiple Instruction, Multiple Data (MIMD) is the most ubiquitous form of parallelism seen today, and it is the category into which most distributed supercomputers fall. The MIMD paradigm represents a grouping of processing elements each of which can process different instructions on multiple data elements concurrently. The MIMD paradigm also includes an important subcategory known as Single Program, Multiple Data (SPMD); in which the tasks that form a single program are explicitly subdivided and run simultaneously on multiple processors.

3.2 Moore's Law

In 1965 Gordon Moore put forth the observation that the number of transistors in dense integrated circuits doubles approximately every two years, a projection we commonly know today as *Moore's Law* [74]. This observation has held true for decades, and has been one of the driving reasons behind increased Central Processing Unit (CPU) performance. These additional transistors allow for on-

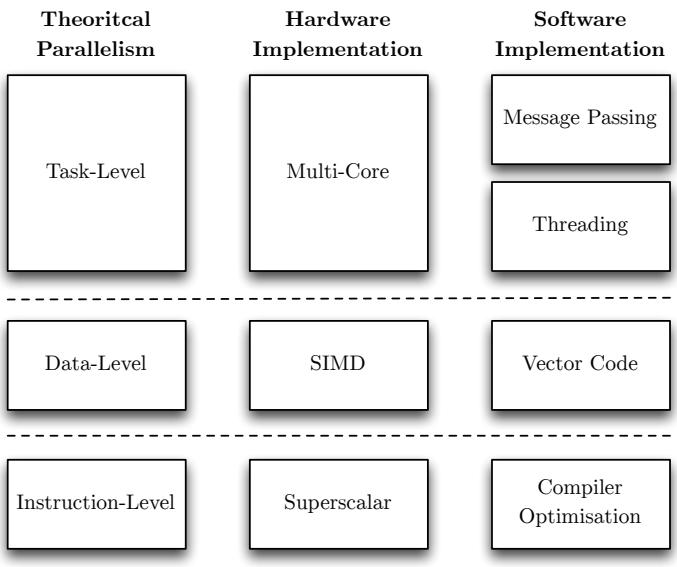


Figure 3.2: Levels of parallelism in theory, hardware, and software.

chip complexity to be added, giving rise to increased ILP, improved cache sizes, and additional arithmetic units.

Despite the continued trend for increasing transistor counts, concerns are growing that this rate of improvement may begin to slow, as transistor size begins to approach the physical limits of current techniques. The increasing consideration for power efficiency means that clock frequencies have also begun to stagnate, with hardware manufactures and consumers alike unwilling to pay the power cost required to drive clock rates further. Now that application developers can no longer rely on this increasing performance driven by Moore's law, a new era of algorithmic research has started. Algorithmic changes are now being recognised as the most important way to drive forward code performance and to get increased utilisation of machine features [29, 63].

3.3 Hardware Parallelism

Figure 3.2 provides an overview of how different types of theoretical parallelism are manifested in hardware and software. It highlights the range of parallelism

available, each level of which needs to be exploited to its fullest in order to achieve optimal performance for a given code.

3.3.1 Instruction-Level Parallelism

ILP is a measure of how many of the operations in a computer program the compute-hardware can perform simultaneously. Historically, ILP has been closely associated with instruction pipelining, however advances in processor design have begun to expose alternative ways to introduce parallelism at the instruction level. This includes techniques such as: super scalar dispatch, speculative execution, and out-of-order execution. Super-scalar dispatch allows multiple instructions to be dispatched to different functional units on a processor concurrently. This is exemplified by processors which employ instruction pipelining, where by multiple instructions are partially overlapped to hide instruction throughput latency. Out-of-order execution is a technique which allows the processor to perform operations in any order that does not violate dependencies, often reducing the effect of stalls caused by unfulfilled data dependencies. Speculative execution allows for the issuing of instructions from different code paths which the processor predicts may be required. This allows the processor to perform computation when it may otherwise be idle. This does, however, come at the risk of the work being wasted should the result of the instruction not being needed.

The introduction of hardware level ILP often comes at the cost of increased CPU design complexity. Techniques such as super-scalar dispatch can often be determined in software by dependency analysis at compile time. This removes the need for dedicated CPU silicon, at the expense of increased compilation and software complexity. This is the approach used by Very Long Instruction Word (VLIW), to introduce increased levels of ILP and to facilitate increased CPU performance without increasing hardware complexity. Doing this allows the transistors saved to be used to improve other areas of functionality.

3.3.2 Vectorisation

Vectorisation is the theoretical concept represented by SIMD processing in Flynn's taxonomy (Section 3.1). A vector processor can be thought of as a CPU which supports an instruction set allowing for the operation of instructions directly on vectors of data (one dimensional arrays).

The idea of vector processing was introduced in the 1970s and was a staple of the supercomputing landscape until the 1990s, after which commodity components, that lacked vector processing capabilities, were widely adopted for use in HPC machines. Current implementations of vectorisation include Streaming SIMD extensions (SSE), and AVX, the utilisation of which is key to achieving high Floating-Point Operations per Second (FLOP/s) performance on the majority of commodity CPU components today. When first introduced, SSE supported the operation of a single instruction on two 64-bit floating point values. AVX has since expanded this to four 64-bit floating point values, with accelerated hardware such as the Intel Xeon Phi supporting 512-bit AVX. As SIMD width increases, so too does the importance of fully utilising it; if a code does not, each time the vector width doubles, 50 percent of the potential performance is lost. In order to achieve maximum performance, programmers often rely on *intrinsic instructions*. These instructions give programmers explicit control over how the hardware's vector registers are used, and which assembly instructions are generated. However, this increased control does not come without cost, as the code generated by programmers in this way is often complex, platform specific, and may not run well on alternative hardware. The code is typically harder to read, and therefore less maintainable than traditional scalar code. For this reason auto-vectorisation is often preferred, whereby the compiler takes scalar code and automatically generates comparable vector code, respecting any data dependencies.

Many of the most important scientific simulations in use today were written with little consideration for vector processing, and subsequently exhibit poor performance on current generations of hardware. This is also true of many

Inertial Confinement Fusion (ICF) codes, and is one of the issues the work in this thesis seeks to address.

3.3.3 Multi-Threading

As hardware manufactures and consumers alike are no longer willing to pay the increasing power costs associated with rising clock frequencies, they are instead opting to favour increasing numbers of cores per processor as a means to achieve greater computational performance. Many-core architectures have decreased per-processor clock-speeds, whilst offering greater performance per Watt. However, this assumption only holds true if programs are able to make effective use of the additional processor cores, which can be accomplished by programming in a multi-threaded capable language. In so doing, a program can use many threads (typically one per processing element) to distribute application workloads, with each thread responsible for some sub-task working towards the global application goal. Many modern architectures also support Simultaneous Multi-Threading (SMT), often referred to as 'hyper-threading'. This allows multiple threads to share the resources of a single physical core, with 2-way SMT being typical on CPUs. SMT is also commonly found on Graphics Processing Unit (GPU) architectures, where the degree of SMT is typically much greater than that of CPU architectures. Historically, such threading was achieved using pthreads, a library which provides a standard way of writing threaded POSIX software [81].

Despite the low level explicit control offered by pthreads, the most prominent way of programming multi-threaded scientific applications, is through the use of OpenMP – a library which abstracts the use and programming of threads. It employs the *fork-join* model (Figure 3.3), where the master thread creates (forks) additional threads which are used to complete parallel code sections, once each thread completes its task, it joins the master thread again before execution is continued. OpenMP is programmed through the use of pragmas, which allow the programmer to mark regions of code with additional information

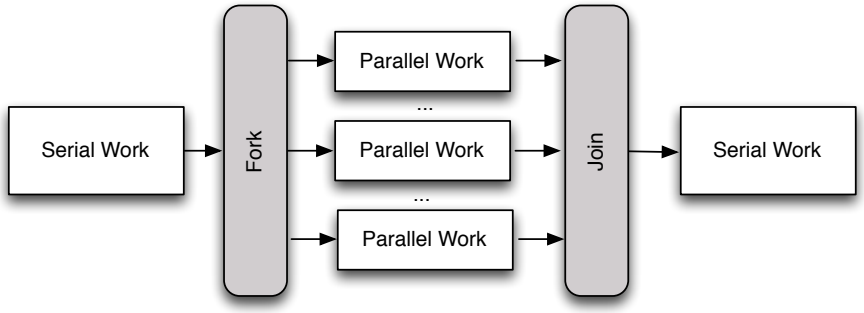


Figure 3.3: Fork-Join Model for Thread-Level Parallelism.

```
1        #pragma omp parallel for schedule(dynamic)
2        for (int i = 0; i < num_particles; i++)
3        {
4          x[i] += ux[i] * weight;
5          y[i] += uy[i] * weight;
6        }
```

Figure 3.4: OpenMP Pragma Example to Distribute Work.

about valid ways to run the program in parallel. An example of such pragma based programming is shown in Figure 3.4. At the start of the loop threads are forked; the loop iterations can then be distributed between the threads, until finally the work has concluded and the threads can join again.

CPUs which feature multiple cores of slower clock speeds come with a variety of advantages, but are not without complication. Typically the CPU chips which exploit the highest degree of multi-threading feature multiple sockets, each with an individual processor containing many cores. These may appear to the operating system as a single processor with many cores, but such multi-socket designs feature regions of Non-Uniform Memory Access (NUMA). This means that any cross-socket (or cross memory-controller) memory accesses pay an increased latency and bandwidth cost, which represents a major performance penalty. Typically programs need to be written and executed in a NUMA aware way to get the best performance from modern hardware and to avoid cache coherency problems such as false sharing. Coherency issues arise when two

threads share write access to the same data region. When a thread writes to a shared cache line, all processors must invalidate this row and re-fetch it from main memory. This can drastically increase memory traffic and, if undetected, can cause a serious performance bottleneck.

Programming models like the Compute Unified Device Architecture (CUDA) and the Open Computing Language (OpenCL) also employ multi-threading by grouping threads together in work-groups. These work-groups have strict guarantees about synchronisation and memory access within a work-group, but offer no guarantees of synchronisation beyond that. Multi-threading in this way shares many similarities with the approach used by OpenMP, but the implementation specific details differ greatly. The use of OpenCL for multi-threading is covered in detail in Section 3.4, and exemplified in Chapter 6 where an account of developing an OpenCL mini-benchmark is presented.

3.3.4 Message Passing

Whilst multi-threaded applications can communicate through shared memory, this is not typically possible for inter-node communication. Instead, communication is usually done through explicit *message passing* via standard libraries (such as the Message Passing Interface (MPI) library). In this paradigm, messages are sent between nodes over a dedicated communication network; it is on top of this that many programming models have been built, including Charm++ [52], HPX [51], Intel's Concurrent Collections (CnC) [60, 61], and COMP Superscalar [106]. These offerings are trying to move programmers away from explicit message passing communication and instead encourage a more conceptual implementation whilst offering functionality such as fault tolerance and data partitioning. While MPI is the dominant standard for current inter-node communication, it is also heavily used for intra-node communication. Despite the advantages of multi-threading, the popularity of message passing for intra-node communication can be attributed to its general ubiquity and high levels of programmer familiarity. Some studies have gone as far as showing that this use

of MPI for intra-node communication is harmful to program runtimes. These performance penalties come from such factors as the inefficient use of the available shared memory, and increased MPI overheads [49, 95]. This has led to the development of *hybrid* applications, which use MPI for inter-node communication, and use OpenMP within a node to fully exploit shared memory. This can be combined with SIMD execution to fully exploit all levels of parallel execution [97].

Message passing is often contrasted with the idea of distributed shared memory; in which physically separated memory architectures can be addressed using single address space, giving access to all available memory on the machine. This means explicit memory passing is no longer needed, as the data sharing between nodes can be done implicitly using the common address space.

3.4 Performance Portability

Performance portability refers to the ideology of having a single application capable of running successfully across a range of hardware architectures, whilst still offering reasonable guarantees about application performance. This issue is becoming increasingly important as commodity hardware designs are begging to diverge and heterogeneous computing is becoming commonplace. Many researchers have demonstrated the importance of fully utilising accelerated architectures, achieving speedups of greater than an order of magnitude [5, 79, 104, 114]. It is likely that any code which is unable to exploit this, will also fail to exploit the increased computational power offered by exascale systems.

Despite the benefits of portable performance being clear, the ideology has not yet been widely adopted. The majority of current offering choose to development a single source code capable of targeting many platforms. Such a solution requires maintenance of only a single codebase, irrespective of the number of platforms being targeted. This means that multiple hand-tuned code versions

are not required, as may have been typical previously. Some tools are beginning to support compilation for wider range of hardware, exemplified by the inclusion of offload support in the OpenMP 4 standard [84].

OpenCL [57] is often thought to be synonymous with portable computing, as it offers a single source solution whose runtime supports the ability to run on a range of hardware. During code execution, work-item and work-group allocation can be tuned at runtime, meaning the granularity of parallelism can be changed to ensure it is suitable for the target hardware. This model of platform agnostic execution avoids a dependency being formed between a codebase and a single hardware technology or vendor. It is this approach that is used for the accelerator work in this thesis, covered in detail in Chapter 6.

3.5 Benchmarking Platforms

The research discussed in this thesis makes use of a wide variety of different hardware and architectures. This Section gives a detailed account of the hardware used and outlines the fundamental differences between them. It was not possible to use the same hardware for every investigation because of limited availability, incompatibility issues between hardware and programming techniques, and the general development of hardware over the time span of this investigation. Whilst performing all experiments on the same hardware may be desirable, this variety strengthens the work and demonstrates the effectiveness of the findings on multiple platforms. All reported experiments were repeated multiple times, with any deviations from the mean being commented on. The data for the graphically presented results in Chapters 6, 7, and 8 can be found in Appendices A, B, and C respectively.

3.5.1 Single Nodes

Here we distinguish between the supercomputers used during experimentation, and the individual hardware components which compose them. Table 3.1 shows

	Intel				
	X5550	**X5650**	**X5660**	**E5-2670**	**E5-2697v2**
Cores	4	6	6	8	12
Clock Speed (GHz)	2.66	2.66	2.80	2.6	2.70
Peak GFLOP/s	42.56	63.984	67.2	332.8	518.4
Bandwidth (GB/s)	32	32	32	51.2	59.7
TDP (Watts)	95	95	95	115	130
Instruction Set	SSE 4.2	SEE 4.2	SSE 4.2	AVX	AVX
Micro-architecture	Nehalem	Nehalem	Nehalem	Sandy Bridge	Ivy Bridge

Table 3.1: Hardware specifications of the CPUs used in this thesis.

the CPUs used in this thesis, and Table 3.2 shows the GPUs used. Throughout the peak Giga-Floating-Point Operations per Second (GFLOP/s) performance for single precision computation is reported, and bandwidth is given as the peak transfer rate in Gigabytes per Second (GB/s) – as quoted by the manufacture. These figures serve as an upper bound for performance, and are rarely achievable during code operation. The power of each machine is reported as Thermal Design Power (TDP) in Watts and the clock speed is reported in Gigahertz (GHz). For the OpenCL work which treats a CPU like a GPU, we can consider the number of cores to be both the number of compute units and processing elements. For the work described in Chapter 6 we used a variety of the CPUs and GPUs described in Tables 3.1 and 3.2 respectively. When comparing performance results across different hardware types, it is essential that every effort is taken to ensure a fair comparison is made. Various works highlight the importance of such fair comparisons, yet no single metric being accepted as the universally correct solution [13, 14, 67, 113]. In instances in this thesis where hardware is directly compared, a combination of age, TDP and cost is used.

3.5.2 Supercomputers

For the work described in Chapter 5 we use two different supercomputers in the validation of the model. Minerva, the resident supercomputer at the University of Warwick; and Sierra, a large scale capability resource located at the Lawrence Livermore National Laboratory (LLNL). The specification of the two machines

	NVIDIA		
	C1060	**C2050**	**K20**
Compute Units	30	14	14
Processing Elements	240	448	2496
Peak GFLOP/s	933	1288	3520
Bandwidth (GB/s)	102	144	208
TDP (Watts)	189	238	225
Micro-architecture	Tesla	Fermi	Kepler

Table 3.2: Hardware specifications of the GPUs used in this thesis.

	Sierra	**Minerva**
Processor	Intel Xeon 5660	Intel Xeon 5650
Cores/Node	12	12
Nodes	1849	258
Memory/Node	24 GB	24 GB
Interconnect	QLogic TrueScale 4X QDR InfiniBand	

Table 3.3: Hardware specifications of *Sierra* and *Minerva*.

used in this study are summarised in Table 3.3. For the work described in Chapter 7 we use ARCHER; a 1.6 Peta-Floating-Point Operations per Second (PFLOP/s) Cray XC30, housed at the UK National Supercomputing Centre at EPCC.

3.6 Summary

In this Chapter we have considered the history of parallel hardware, how it is classified, and how it has evolved. We present a summary of the key features needed to understand the work in this thesis, with a strong emphasis on hardware parallelism. Exploiting this parallelism is one of the major challenges faced

	Archer (Cray XC30)
Processor	Intel Xeon E5-2697v2
Cores/Node	24
Nodes	4920
Memory/Node	64 GB
Interconnect	Cray Aries Interconnect

Table 3.4: Hardware specifications of *ARCHER*.

by HPC, with many current codes failing to fully exploit either multi-threading or vectorisation – instead solely employing message passing paradigms to utilise a subset of the parallel computing topology. In the remainder of this thesis we build on our knowledge of compute hardware and ICF simulation to develop techniques to help address these issues, with a particular focus on Magnetohydrodynamics (MHD) and Particle-in-Cell (PIC) simulations.

CHAPTER 4

Inertial Confinement Fusion Simulations

Whilst the work detailed in this thesis applies to a range of areas of Inertial Confinement Fusion (ICF) research, it focuses most strongly on Particle-in-Cell (PIC) and Magnetohydrodynamics (MHD) simulations. Many of the techniques described are applicable to other simulation types, as demonstrated by the performance model in Chapter 5. The work in Chapter 6 and Chapter 7 focuses more specifically on PIC codes, but is representative of a wide range of codes.

4.1 Motivation

In September 2013, the large laser-based inertial confinement fusion device housed in the National Ignition Facility at the Lawrence Livermore National Laboratory (LLNL), was widely acclaimed to have achieved a milestone in controlled fusion – successfully initiating a reaction that resulted in the release of more energy than the fuel absorbed. Despite this success, the ICF community remains some distance from being able to create controlled, self-sustaining, fusion reactions. ICF represents one leading design for the generation of energy by nuclear fusion. Since the 1970s ICF has been supported by computer simulations, providing the mathematical foundations for pulse shaping, lasers, and material shells needed to ensure effective and efficient implosion.

The UK has a long history of research into high-intensity laser plasma interactions. The UK's Central Laser Facility is home to some of the world's most advanced high power lasers, which can deliver petawatt focused beams, with approximately 10,000 times more power than the UK National Grid during picosecond pulses. Developments in the deployment of relativistically intense

'long' laser pulses (to compress fuel) and fast 'short' pulses (for ignition) present significant challenges in computational plasma physics. Plasmas with intense electromagnetic fields require fully kinetic models of particle distribution in seven dimensions (three representing space, three representing momentum and one representing time); and point design for targets requires the coupling of relativistic kinetic models with long time-scale radiation hydrodynamics codes. As future fusion codes continue to develop to support plasma turbulence studies, in order to exploit facilities such as ITER and the National Ignition Facility (NIF), so too does the complexity of these simulations and the increasing demands on the supporting supercomputers. Any effort which improves code performance is valuable; as not only will it facilitate faster scientific discovery, but it will also aid efficient use of both experimental laser facilities and simulation equipment.

This Chapter presents a summary of two such simulation codes used extensively in the UK's ICF efforts, and throughout this thesis: EPOCH, a fully relativistic particle-in-cell plasma physics code, developed by a leading network of over 30 UK researchers; and Lare, a leading UK MHD code. A significant challenge in developing large codes is maintaining effective scientific delivery on successive generations of high-performance computing architectures. To support this process, the use of mini-applications, mini-benchmarks, and performance models is adopted.

4.2 Background

This Section provides a background to scientific simulations in general, and the specific grid based codes used in this thesis. Both Lare and EPOCH use an n-dimensional spatial grid to track discretised quantities (such as magnetic fields). These groupings of points can then be treated as vector fields if required. The typical way of describing such grids is to track their size in grid cells per dimension, denoted as N_d, where d represents a given dimension in a two or three dimensional space (*e.g.* N_x to represent the x dimension). Each grid cell,

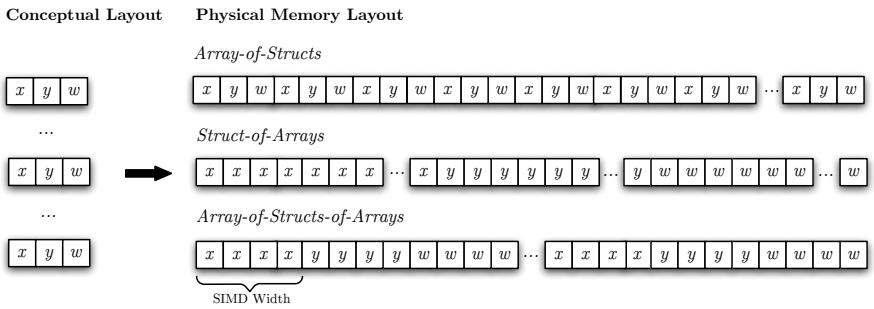


Figure 4.1: A comparison of Array-of-Structs, Struct-of-Arrays, and Array-of-Structs-of-Arrays data layouts.

typically of type `double`, represents a configurable region of real space, with the size and number of grid cells being a runtime parameter which determines the amount of work the simulation will undertake and the physical domain being simulated. These codes are run iteratively for a fixed number of timesteps, after which the simulation completes.

Such simulations are decomposed to run on N processing elements by employing a spatial decomposition, most typically using the Message Passing Interface (MPI) library. During the decomposition, the physical domain is initially split between the processors to give even workloads using an algorithm which minimises the surface-area to volume ratio, as the size of the surface area is typically strongly correlated with the scale of inter-process communication. Extra cells of storage are then added to each edge of these decomposed regions, in which temporary data from neighbouring regions can be stored. This is a common technique in scientific simulation, and is known as storing *ghost cells* (or *halo cells*). These ghost cells can then be exchanged during communication phases to resolve any data dependencies.

One consideration for such simulations is the way in which they store the data required for their operation. Many legacy simulations rely heavily on linked lists to store data, especially in instances where the data ordering frequently changes, as linked lists provide a good conceptual fit offering easy insertion and deletion. While the use of a linked list does not necessarily present a problem, a naïve

implementation of a linked list may offer no guarantees of contiguous memory accesses, and often lead to a fragmented memory space. This fragmentation can significantly impact code performance, as modern hardware is optimised for contiguous memory loads, with each issued memory load fetching an entire cache line. If data is not tightly packed in memory, non-required data will be brought in from main memory – thereby wasting memory bandwidth.

The most common way to avoid this problem, is to use array-based data stores. As an array has strong guarantees on the memory-use being contiguous, fragmentation is unable to occur in the same way. Typically, this means storing an array of custom objects (or structs), a technique known as an *Array-of-Structs (AoS)*. By storing program data as an AoS, a single memory stream is required by the prefetcher, as well as providing a simplified programming interface. This may, however, come at the cost of decreased Single Instruction, Multiple Data (SIMD) efficiency; when processing data from multiple objects in AoS, memory accesses will be strided causing non contiguous memory accesses. Two alternative approaches to array-based storage include a *Struct-of-Arrays (SoA)* and a more complex hybrid, *Array-of-Structs-of-Arrays (AoSoA))* (which aims to combine the benefits of both SoA and AoS). A brief overview of these memory layouts is found in Figure 4.1. For the SoA data layout, single properties for multiple data elements are stored together in an array. This means that under SIMD operation, single properties from multiple elements can be loaded in one contiguous and aligned load, at the expense of tracking a different memory stream per property required. This eliminates any potential for gather/scatters, and is often favourable when only a few data properties are required. With AoSoA, groups of N elements of each property are stored together, in order, where N is typically a function of vector length. This approach attempts to combine the benefits of both SoA and AoS with aligned loads and few memory streams, but comes at the expense of vastly increased complexity and an indexing overhead.

4.3 Lare

Lare is a multidimensional MHD code, the core solver of which focuses on solving the idealistic MHD equations (given in Equations 4.1–4.4) using a predictor-corrector scheme based on control volume averaging. This scheme uses a staggered Eulerian grid in order to accurately reproduce the shocks found in MHD reactions. The equations which define MHD describe the relationship between electrically conducting fluids and the effects they have on surrounding magnetic fields. In these equations the magnetic field is represented by $\boldsymbol{B}$, the mass density is represented by ρ, velocity is represented by v, thermal pressure is represented by P, resistivity is represented by η, and the specific heat ratio (with a typical value of $5/3$) is represented by γ.

$$\frac{D\rho}{Dt} = -\rho\nabla\cdot\boldsymbol{v} \tag{4.1}$$

$$\frac{D\boldsymbol{v}}{Dt} = \frac{1}{\rho}(\nabla\times\boldsymbol{B})\times\boldsymbol{B} - \frac{1}{\rho}\nabla P \tag{4.2}$$

$$\frac{D\boldsymbol{B}}{Dt} = (\boldsymbol{B}\cdot\nabla)\boldsymbol{v} - \boldsymbol{B}(\nabla\cdot\boldsymbol{v}) - \nabla\times(\eta\nabla\times\boldsymbol{B}) \tag{4.3}$$

$$\frac{D\varepsilon}{Dt} = -\frac{P}{\rho}\nabla\cdot\boldsymbol{v} + \frac{\eta}{\rho}j^2 \tag{4.4}$$

Additionally, the current density (j) and internal energy (ε) are defined as:

$$j = \nabla\times\boldsymbol{B} \tag{4.5}$$

$$\varepsilon = \frac{P}{\rho(\gamma-1)} \tag{4.6}$$

In solving these equations, Lare first takes a second-order accurate *Lagrangian step* to update the velocities, the magnetic fields, and the artificial viscosities; a process performed by evaluating derivatives on the original grid.

An additional step is then required to remap the properties back to the Eulerian grid. This step is known as the original *Eulerian Remap*, during which a series of one-dimensional sweeps map the system properties back onto the

```
1        DO
2        IF (step .GT. n_steps) EXIT
3        ...
4        CALL set_dt
5        CALL lagrangian_step
6        CALL eulerian_remap(i)
7        CALL boundaries
8        ...
9        END DO
```

Figure 4.2: The main compute loop of Lare, operated over for a fixed number of iterations.

original Eulerian grid using a van Leer [112] piecewise linear reconstruction. In so doing, this remap fully conserves mass, internal energy, and momentum – ensuring simulation accuracy is maintained.

This method shares many similarities with Arbitrary Lagrangian-Eulerian (ALE) codes, a technique used in both ICF research, as well as other fields such as hydrodynamics and finite volume simulation. ALE attempts to offer the benefits of both Lagrangian and Eulerian schemes, typically allowing for mesh movement to be of arbitrary velocity. Like the scheme used in Lare, ALE codes typically split the time update into a Lagrangian step to advance the simulation, and then later remap the grid. Unlike the scheme used in Lare however, this can be to an arbitrary grid and does not have to be done every time step; instead the remap can be deferred until the grid is deemed to be sufficiently distorted. An ALE extension to Lare is currently under development, henceforth referred to as *ODIN*. The development of ODIN would allow the costly remap step to be done less frequently, whilst also expanding the set of physical simulations the two codes are able to reproduce. Some initial performance predictions for the operation of ODIN are presented in Chapter 5.

4.3.1 Computational Overview

As with all computational simulation, it is highly important to gain a good understanding of their operation to ensure all decisions and changes are well informed. In addition to understanding the physics used in Lare, this Section also

Subroutine	Percent Runtime
remap_y	27.81
remap_x	22.33
lagrangian_step	18.89
energy_account	15.94
set_dt	8.38
eulerian_remap	1.50
remap_z	0.75
other	4.40

Table 4.1: A profile of Lare runtime composition.

provides the information needed to gain an understanding of how Lare is expressed as code, and how it operates. In so doing we can gain valuable insight into how it may perform on current and future generations of compute hardware, as well as understand how certain changes may affect its operation.

The main loop of Lare is shown in Figure 4.2, and highlights the *Lagrangian step* and the *Eulerian remap* at the heart of the algorithm which, when combined, dominate the application runtime. During code operation, a fixed size grid of size $N_x \times N_y$ is run for a given number of iterations. As previously described in Section 4.2 and as is typical with most grid based simulations, work is distributed through an MPI domain decomposition to ensure each processing element has a comparable workload.

As may be expected, the Lagrangian step contains the majority of the computationally intensive physics, and subsequently representing a significant proportion of the runtime and the majority of the floating point operations. The *Eulerian remap* also involves a significant amount of computation, data movement and a series of near-neighbour exchanges needed to ensure neighbouring cells hold the appropriate values. Table 4.1 shows typical runtimes for each Lare sub-component, with 34.83% of the time being attributed to Lagrangian step (composed of lagrangian_step and energy_account), and 52.39% attributed to the Eulerian remap (composed of remap_x, remap_y, remap_z and eulerian_remap). This timing data gives us insight into how changes to different code segments may influence the overall application runtime.

In Chapter 5 we build on this knowledge of Lare and present the development of a performance model capable of accurately predicting both compute and communication times to greater than 90 percent accuracy, as well as making predictions about the runtime performance of the ALE based code variant, ODIN. Not only does this work highlight the importance of code understanding for ICF simulations, but also underlines the power of performance modelling.

4.4 EPOCH: Extendable PIC Open Collaboration

EPOCH [4] is a nationally funded, fully relativistic Electromagnetic PIC plasma physics code, developed by a network of over 30 UK researchers. It is the UK's leading PIC code, a class of codes which are amongst the most widely used computational tools in plasma physics research, and are crucial for further understanding of both ICF and the field of laser-plasma interactions in general.

At the core of the EPOCH codebase are particle push and field update algorithms developed by Hartmut Ruhl, first seen in the PSC code [100]. These have then been extended to include advanced features such as particle collisions, ionisation and Quantum Electrodynamics (QED) driven coherent radiation. During code operation, EPOCH tracks the electric and magnetic fields generated by the motion of (pseudo)particles, and is capable of reproducing the full range of classical microscale behaviour required to accurately simulate a collection of charged particles. To do this, EPOCH uses the Finite-Difference Time-Domain (FDTD) method to solve Maxwell's equations numerically. This scheme uses a modified leapfrog method, in which the field is updated at both the full timesteps (n), and the half timestep ($n + \frac{1}{2}$). Initially, the magnetic fields ($\boldsymbol{B}$) and electric fields ($\boldsymbol{E}$) are advanced to the half timestep by using currents calculated at timestep n. Once the particles have been pushed (Equation 4.11), the fields can then be updated to the full timestep (Equations 4.7–4.10). During this field update the electrical current is represented by $\boldsymbol{J}$, and the speed of light is represented by

c. Δt is the Courant-Friedrichs-Lewy (CFL) limited time-step used to advance the simulation [22].

$$\mathbf{E}^{n+\frac{1}{2}} = \mathbf{E}^{n} + \frac{\Delta t}{2}\left(c^2\nabla\times\mathbf{B}^{n} - \frac{\mathbf{J}^{n}}{\epsilon_0}\right) \tag{4.7}$$

$$\mathbf{B}^{n+\frac{1}{2}} = \mathbf{B}^{n} - \frac{\Delta t}{2}\left(\nabla\times\mathbf{E}^{n+\frac{1}{2}}\right) \tag{4.8}$$

$$\mathbf{B}^{n+1} = \mathbf{B}^{n+\frac{1}{2}} - \frac{\Delta t}{2}\left(\nabla\times\mathbf{E}^{n+\frac{1}{2}}\right) \tag{4.9}$$

$$\mathbf{E}^{n+1} = \mathbf{E}^{n+\frac{1}{2}} + \frac{\Delta t}{2}\left(c^2\nabla\times\mathbf{B}^{n+1} - \frac{\mathbf{J}^{n+1}}{\epsilon_0}\right) \tag{4.10}$$

In order to push the particles, the particle pusher solves the relativistic equation of motion under the Lorentz force [69] for each particle in the simulation. The particle trajectory is calculated to second order accuracy using the electric and magnetic fields at the half time-step (as above). Each particle is then updated (pushed) according to:

$$\boldsymbol{p}_i^{n+1} = \boldsymbol{p}^{n} + q\Delta t\left[\mathbf{E}^{n+\frac{1}{2}}(\boldsymbol{x}^{n+\frac{1}{2}} + \boldsymbol{v}^{n+\frac{1}{2}}\times\mathbf{B}^{n+\frac{1}{2}}(\boldsymbol{x}^{n+\frac{1}{2}})\right] \tag{4.11}$$

Where $\boldsymbol{p}$ is the particle momentum, $\boldsymbol{q}$ is the particle charge, $\boldsymbol{x}$ is the particle position, and $\boldsymbol{v}$ is the particle velocity.

4.4.1 Computational Overview

Conceptually, the collisionless PIC algorithm implemented in EPOCH can be thought of as performing the following four discrete steps (shown as pseudocode in Figure 4.3):

1. Move the particle across the physical domain, proportional to particle momentum;

2. Update the particle's momentum, based upon the local electric fields, magnetic fields, and particle shape;

3. Deposit the generated current onto the grid, to act as an intermediary for

```
for all species do
    for all particles do

        ▷ Move particles.
        position ← position + momentum

        ▷ Update momentum based on field effects.
        e_cell ← ⌊position⌉
        for all neighbours of e_cell do
            calculate electric field effects
        end for

        b_cell ← ⌊position + 0.5⌋
        for all neighbours of b_cell do
            calculate magnetic field effects
        end for

        momentum ← momentum + electric and magnetic field effects

        ▷ Calculate and deposit currents.
        for all neighbour cells do
            calculate current
            deposit current
        end for

    end for
end for
```

Figure 4.3: Pseudocode depiction of EPOCH's core PIC algorithm.

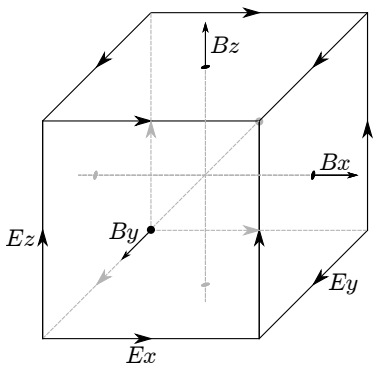


Figure 4.4: An example of a Yee staggered grid.

particle-particle interactions;

4. Update the electric and magnetic fields according to the current deposited.

Using this approach it is possible to reproduce the full range of classical micro-scale behaviours of a collection of charged particles. Like many codes of this type, EPOCH is Fortran-based and parallelised using MPI. Dynamic load balancing options exist and MPI-IO allows checkpoint restart on an arbitrary number of processors. Legacy simulation codes designed and implemented in this way are now exhibiting poor utilisation of modern hardware features such as vector operations, and fail to fully exploit all levels of available parallelism – a problem which is exacerbated by the energy-efficient benefits available through heterogeneous computing. In EPOCH, particles are densely packed, with a single particle spanning multiple grid cells (typically 3×3). EPOCH represents electrical and magnetic fields on a staggered Yee grid [116], as shown in Figure 4.4. During the momentum update (step 2), a 25-point stencil in each of 3 dimensions is read per field, and used to update the particle momentum; the scale of these memory operations are a significant contributor to the overall application runtime. Once the current contributions have been calculated (step 3), they are then stored into a global array at indices determined by the grid vertices touched during particle movement. This write to multiple indices of a global array limits the possibility of particles simultaneously depositing current without the need for atomics or other concurrency control. It is this *current deposition* that most strongly differentiates PIC from alternative methods; unlike molecular dynamics, for example, PIC features no particle-particle interactions, instead approximating these using the Yee grid as an intermediary.

Table 4.2 shows the distribution of runtimes for a typical EPOCH run. We can see that the large kernel which represents the core push algorithm dominates the runtime, as this includes: the movement of the particles (step one); the update of the particles (step two); and the current deposition (step three). The next biggest contributor is the field updates, which represents the final step of

Code Segment	Percent Runtime
Particle Push	81.63
Electric / Magnetic Field Update	11.49
Boundary Conditions	4.02
Other	2.86

Table 4.2: A profile of EPOCH runtime composition.

the algorithm.

Finally, as point of clarification, a range of PIC codes exist, and the work presented here most closely applies to fully relativistic PIC codes such as EPOCH, where Maxwell's equations are solved using FDTD methods. Other classes of code include those which utilise gyrokinetic equations, or those which employ Fourier transforms to operate in the time domain. Whilst many of the findings presented here may apply to these, no special consideration is made for them.

4.5 Summary

This Chapter offers some background into the importance of ICF simulation codes and their operation. It shows the importance of the UK's role in both ICF research and in shaping the face of renewable energy. We present an overview of the relevant simulations (Lare and EPOCH), and give a background to how such simulations typically operate. We detail the governing equations, and give a conceptual overview of their implications with a focus on how this affects the codes computational performance. A key theme throughout this thesis is the importance of understanding code performance, and this Chapter hopes to give an insight for the codes in question. By applying the knowledge of the codes presented here, combined with the theory laid out in Chapters 2 and 3, a good understanding of code performance, and changes needed to improve it can be gained.

CHAPTER 5

Performance Modelling of Magnetohydrodynamics Simulations

Performance modelling is the prediction of code runtime based on inferences about the behaviour of the application, and the performance characteristics of the underlying computing system. This concept becomes more important with the increasing complexity of modern architectures, and the rise of heterogeneous computing. It is important to ensure that such resources are used effectively, and given the highly parallel Single Instruction, Multiple Data (SIMD) nature of many-core units, such as Graphics Processing Units (GPUs) and the Intel Xeon Phi product line, this may not be a straight forward task [38, 85, 101]. By being able to accurately predict the runtime of a code for a given architecture, it is not only possible to make more efficient use of the hardware, but code performance can also be rapidly compared for a variety of different architectures. Furthermore, results can be extrapolated past existing core counts to make predictions and to reason about code performance at scale. In order to predict runtime performance, application behaviour must be analysed and the performance characteristics of the target system must be well understood.

This Chapter describes the development of a performance model for the 2-dimensional variant of Lare, a representative plasma physics application used in Inertial Confinement Fusion (ICF) research. Performance models are a vital tool used by the High Performance Computing (HPC) community in order to predict the runtime of an application. These predictions can then be used to aid procurement decisions, identify optimisation opportunities, or to predict the behaviour of an application running on a hypothetical future architecture at scale [56].

5.1 Development of a Performance Model

We directly build upon the process discussed in Section 2.5 to develop a performance model capable of fully representing Lare. In order to fully understand the runtime characteristics of Lare, the code was profiled for both serial and parallel runs. This quantifies the time spent in each subroutine, and provides a metric which guides the appropriate level of effort for each code section during the creation of the simulation. To construct a model using SST/macro, a skeleton of the code that includes the main areas of compute and communication has to be constructed. As the generation of a comprehensive skeleton application can be a non-trivial process, a small tool was written to facilitate this. The tool performs static analysis on the Fortran source code, and transforms this information into a SST/macro skeleton model. The tool parses the Fortran source code line by line, splitting the line into tokens based on whitespace. These tokens are then matched against an in-built list of keywords, identifying areas such as subroutine declarations and invocations. Once a keyword is matched, the line is processed and added to the model. Subroutine declarations are parsed and replicated in the skeleton code. These subroutines are then populated by any function calls made within them. One of the key benefits of the tool is that it identifies Message Passing Interface (MPI) communications and is able to flag these to the user and input them into the skeleton. The tool is able to auto-complete much of the information about the MPI call needed by the SST/macro API, leaving only the size of the communication buffer to be provided by the user. In addition to the skeleton, SST/macro requires machine specific details to be specified, such as: topology, network bandwidth, and on and off-node latencies. In order to accurately populate the skeleton application, the main contributors of runtime need to be identified. By profiling Lare and combining this with our existing understanding, it is clear that the two most significant contributors are the Lagrangian step and the Lagrangian remap (as previously discussed in Table 4.1). By combining these two steps, an equation that accurately and

File Name	Subroutine	w_g Term
diagnostics.f90	energy_account	$w_{energy\_account}$
lagran.f90	lagrangian_step	$w_{lagrangian\_step}$
lagran.f90	predictor_corrector_step	$w_{predictor\_corrector}$
xremap.f90	remap_x	w_{remap_x}
yremap.f90	remap_y	w_{remap_y}
zremap.f90	remap_z	w_{remap_z}
remap.f90	eulerian_remap	$w_{remap\_remainder}$
diagnostics.f90	set_dt	w_{set_dt}

Table 5.1: The grind times used in modeling Lare, including their relative location in the source code.

concisely summarises the total runtime of Lare can be developed, as shown in Equation 5.1.

$$T_{total} = \sum_{i=0}^{iterations} (t_{lagrangian\_step} + t_{remap}) \tag{5.1}$$

In order to make use of this equation, an incremental approach to building a model was taken, starting with the construction of a serial model.

5.1.1 Serial Model

For a serial run of Lare, there is no inter-process communication – the runtime is singularly representative of the compute, allowing us to apply Equation 5.2, which can be derived from Equation 2.6 (Section 2.5) by reducing the T_{comms} term to 0.

$$T_{total} = T_{compute} \tag{5.2}$$

This equation can be decomposed further. The term $T_{compute}$ can further be broken up into its subcomponents, as shown in Equation 2.7. A table listing the relevant grind times for Lare can be found in Table 5.1. The relevant w_g times can be derived by running a version of Lare instrumented with timers. Using these values, a model can be developed which is capable of predicting serial runtime to an exceptionally high level of accuracy, using only Equation 5.1.

5.1.2 Parallel Model

Once a serial model has been developed, a parallel model can then be considered in the form shown in Equation 2.6. Lare's communication is dominated by two MPI function types: `send-receives`, and `all reduces`. The send-receive functions are used to swap neighbour cells, whilst the reduction operations collate data. By taking the sum of the time taken by these operations, the communications time can be represented as:

$$T_{comms} = \sum t_{Sendrecv} + \sum t_{Allreduce} \tag{5.3}$$

During the point-to-point communications, the amount of data sent is dependent on the grid size set at compile time. The grid undergoes a coarse decomposition in two dimensions, and is distributed among the processors. This method of decomposition is performed with the aim of minimising the surface-area-to-volume ratio, which in turn increases the ratio of computation to communication. This decomposition strategy is replicated in the model by an explicit surface-area-to-volume calculation, with SST/macro simulating an exact copy of the communications. Once all the required terms have been identified, they can be incorporated into the model. In order for SST/macro to accurately simulate communications, it requires values for the latency and bandwidth of the target system. These values can be found experimentally with a set of micro-benchmarks that are distributed with SST/macro.

Figure 5.1 shows elements of both the model and original Lare source code for two methods, `dm_x_bcs` and `remap_x`. It compares the original source to the equivalent representation in the model. In Figure 5.1 (a) we see the `dm_x_bcs` subroutine that features an *MPI_Sendrecv*. In (b) we can see this has been translated to the equivalent SST/macro MPI call, to be dealt with by the simulated network. Similarly (c) shows an area of compute performed by the original source, which is then replaced by a grind time (w_g) based calculation in (d) (represented as `compute(t)` in Figure 5.1).

(a) Original Fortran `dm_x_bcs` Subroutine

```
1  SUBROUTINE dm_x_bcs
2      ...
3     CALL MPI_SENDRECV(dm(nx-1, 0:ny+1), ny+2, mpireal, &
4          proc_x_max, tag, dm(-1, 0:ny+1), ny+2, mpireal, &
5          proc_x_min, tag, comm, status, errcode)
6      ...
7  END SUBROUTINE dm_x_bcs
```

(b) Model `dm_x_bcs` Subroutine

```
1  void dm_x_bcs(int rank) {
2      ...
3      mpi->sendrecv(ny + 2, sstmac::sw::mpitype::mpi_real, \
4      proc_x_max, tag, ny + 2, sstmac::sw::mpitype::mpi_real, \
5      proc_x_min, tag, world(), stat);
6      ...
7  }
```

(c) Original Fortran `remap_x` Subroutine

```
1  SUBROUTINE remap_x ! remap onto original Eulerian grid
2      ...
3      DO iy = -1, ny+2
4          iym = iy - 1
5          DO ix = -1, nx+2
6              ixm = ix - 1
7              ...
8          END DO
9
10     END DO
11     ...
12 END SUBROUTINE remap_x
```

(d) Model `remap_x` Subroutine

```
1  void remap_x(int rank) {
2      ...
3      sstmac::timestamp t(remap_x_w * nx * ny);
4      compute(t);
5      ...
6  }
```

Figure 5.1: Code examples comparing original source code with its representation in the model, including a w_g based compute call and a SST/macro MPI call.

5.2 Validation

In order to validate the model, application runtimes are compared with the predicted simulation runtimes across a variety of grid sizes and processor counts on 2 different machines: Minerva and Sierra; the specifications of which are summarised in Section 3.5. For Sierra, version 12.0 of the Intel compiler was used in conjunction with MVAPICH2 version 1.7. For Minerva, version 12.0 of the Intel compiler was used in conjunction with OpenMPI 1.4.3.

5.2.1 Weak Scaled Problem

In the practice of weak scaling, the grid size is increased with the processor count with the aim of keeping the compute-per-processor cost fixed. This is the approach taken for solving increasingly difficult problems in a fixed amount of time. As the processor count increases, more communication between grid cells is required, leading to a general increase in communication time. As the compute per processor remains the same throughout, it can be expected that the w_g values will not change, allowing us to be confident of the predictions for compute time.

Table 5.2 presents a comparison of the experimental runtimes against predicated run-times for a weak scaled problem with 3,000,000 cells per core, running for 100 iterations. The table shows that the model is able to accurately predict the runtime to an accuracy of greater than 90%. The predicted runtime being consistently slightly lower than the experimental time can be attributed to a small percentage of the runtime behaviour not being incorporated in the prediction, such as the set up costs, which are not captured by the model. As the processor counts increase, it is possible that the model may begin to over-predict, but for reasonable node count the current model exhibits sufficient accuracy. The accuracy of the models presented in this Chaptercould be further improved by the inclusion of all elements from the parent code, as well a more detailed exploration of values used to prime the models.

Nodes	Grid Size	Time (s)	Prediction (s)	Error (%)
1	6000	543.10	527.03	-3.05
4	12000	554.90	528.57	-4.98
9	18000	560.63	541.55	-3.52
16	24000	569.41	549.06	-3.71
21	30000	570.08	551.14	-3.44
36	36000	578.24	558.15	-3.60

(a) Minerva

Nodes	Grid Size	Time (s)	Prediction (s)	Error (%)
1	6000	480.70	465.46	-3.29
4	12000	485.26	466.17	-4.10
9	18000	493.59	466.83	-5.73
16	24000	498.32	476.30	-4.62
21	30000	499.07	478.43	-4.31
36	36000	499.01	480.49	-3.85
49	42000	499.47	481.98	-3.63
64	48000	499.15	483.68	-3.20
81	54000	499.31	487.22	-2.48
100	60000	499.58	488.59	-2.25
121	66000	500.00	490.12	-2.02
144	72000	500.57	491.54	-1.84
169	78000	500.29	492.91	-1.50
196	84000	500.27	495.44	-0.98
225	90000	500.85	496.88	-0.80
256	96000	500.29	499.44	-0.17

(b) Sierra

Table 5.2: A comparison of the runtimes and simulation times of Lare on (a) Minerva and (b) Sierra for a weak scaled problem.

5.2.2 Strong Scaled Problem

Strong scaling describes the process of solving a fixed problem size with an increasing number of processors. As the processor count increases the aim is to decrease the runtime. A comparison between experimental runtime and predicted runtime is shown in Table 5.3 for a 16,800 $\times$ 16,800 strong scaled problem, running for 100 iterations. This problem size was chosen to give a sufficiently long runtime, but still fit in the available memory.

Nodes	Time (s)	Prediction (s)	Error (%)
8	518.01	532.85	2.78
12	348.16	364.61	4.51
16	262.74	277.77	5.41
24	172.01	189.51	9.24
32	128.67	133.48	3.61

(a) Minerva

Nodes	Time (s)	Prediction (s)	Error (%)
16	251.06	236.00	-6.38
32	119.60	121.78	1.79
64	61.02	64.16	4.90
128	33.38	35.55	6.12

(b) Sierra

Table 5.3: A comparison of the runtimes and simulation times for Lare on (a) Minerva and (b) Sierra for a strong scaled problem.

For all problems sizes, across both weak and strong scaling, the performance model is able to predict the runtime to an accuracy of greater than 90% for a range of core counts. This is a significant contribution which gives us confidence in our model when using it to make further predictions.

5.3 Evaluation of Future Optimisations

An Arbitrary Lagrangian-Eulerian (ALE) generalisation of Lare is currently under development (ODIN). This would mean the requirement to remap each iteration will no longer hold, and instead a move to an ALE method would allow the remap step to only be done once the grid becomes sufficiently deformed. By performing an investigation into the expected performance of a hypothetical ALE variant of Lare, a valuable insight into the potential performance can be gained.

By moving to an ALE code, the frequency of the remap can now be varied. A metric could be developed to formally determine the optimum value of this frequency (F_r), but initial indications show that remapping will be required, on

average, once every tenth iteration ($F_r = 0.1$) over the course of the simulation. By varying the frequency of the remap, the code will be affected in two main ways. Firstly, it will significantly reduce the general cost per iteration in terms of compute, as the remap step will no longer be present. Secondly, reducing the frequency of the remap step reduces the frequency of inter-process communication. In changing the code in this way, the total cost is no longer as described in Equation 5.1, but instead includes a term to denote the new remap, as in Equation 5.4.

$$T_{total} = T_{lagrangian\_step} + T_{remap\_new} \tag{5.4}$$

This equation can then be reduced further, as shown in Equation 5.5.

$$T_{total} = \sum_{i=0}^{iterations} (t_{lagrangian\_step}) + \sum_{j=0}^{iterations*F_r} t_{remap\_new} \tag{5.5}$$

In order to express the new total cost, relative to the old, Equation 5.1 can be extended to include terms for the relative costs, as shown in Equation 5.6.

$$\begin{aligned} T_{total\_new} = &(T_{lagrangian\_step} \times C_{lagrangian\_step}) + \\ &(T_{lagrangian\_remap} \times C_{remap\_new} \times F_r) \end{aligned} \tag{5.6}$$

If no change to the cost of the Lagrangian step ($C_{lagrangian\_step} = 1$) is assumed, an investigation into how the frequency of remap and the cost of remap affect the overall performance can be performed. Table 5.4 shows the percentage decrease in runtime obtained for different values of F_r and C_{remap_new} for an 8,192 × 8,192 problem on 36 processors performing 100 iterations, in which the remap step contributes just under 65% of the runtime.

From Table 5.4 it can clearly be seen that reducing the remap frequency offers large performance gains as the remap frequency decreases for reasonable values of C_{remap_new}. Optimistic projections for this optimised code suggest

C_{remap_new} \ F_r	**1**	**0.5**	**0.25**	**0.2**	**0.1**	**0.001**
1	0.00	32.15	48.22	51.44	57.87	64.24
2	-64.30	0.00	32.15	38.58	51.44	64.17
4	-192.90	-64.30	0.00	12.86	38.58	64.04
5	-257.20	-96.45	-16.07	0.00	32.15	63.98
10	-578.69	-257.20	-96.45	-64.30	0.00	63.66

Table 5.4: Percent decrease in runtime for different values of F_r and C_{remap_new} for a 8,192 square problem on 36 processors performing 100 iterations.

that it will have a similar cost for the lagrangian step ($C_{lagrangian\_step} = 1$), a remap cost that is around twice as large ($C_{remap\_new} = 2$) and allow the remap to be performed on average every ten steps ($F_r = 0.1$). Table 5.4 shows that this may offer a speedup greater than 50%.

5.4 Summary

This Chapter presented a predictive performance model for Lare, a Magnetohydrodynamics (MHD) code developed by, and maintained at, the University of Warwick. This model allows for the accurate prediction of Lare's runtime on a variety of platforms; it has been demonstrated to be accurate to within 10% of the observed runtime on two clusters, a commodity cluster located at the University of Warwick and a 360 TFLOP/s capability resource located at the Lawrence Livermore National Laboratory (LLNL). The model was shown to perform well for both weak and strong scaling over a wide range of core counts. The model has also been used to provide a forward look at possible optimisations in the Lare code base, with an evaluation of the gains that may be expected. This model could easily be extended to cover the 3-dimensional variant of Lare, with the techniques used being applicable to all such similar codes.

A key theme for this Chapter is understanding the operation of a code and the ability to determine how different hardware architectures will affect this. As the range of diverse hardware continues to increase, it is important to un-

derstand the ways which code can make efficient and effective use of it. An increasing trend in HPC research is the need for portable performance, something which is nearly impossible without a solid understanding of both the code and how it is likely to map to the underlying hardware.

CHAPTER 6

Performance-Portable Plasma Physics Simulations

The plasma physics applications required by many High Performance Computing (HPC) centres are expensive to produce, in part due to the highly complex nature of the mathematics involved, as well as the desired ability to support a wide range of simulations within a single software package. Developing such an application (a process which can be spread over many years, or even decades) also represents a significant investment, often including the creation of novel algorithms which must be rigorously tested by domain experts and computer scientists before the application can be deployed into a production environment. It is therefore important to such centres that codes remain usable for as long as possible, to maximise the return on their investment.

Many such codes were written to target traditional HPC clusters comprised of a large number of high-performance serial cores connected via some networking interface. As such they have been engineered to ensure efficient inter-node message passing patterns and scaling behaviours, but are not likely to be well prepared for the many forms of on-node parallelism that typify the most recent and future architecture designs [27]. It is therefore crucial that codes are revisited to ensure both: portability between the different architectures available today; and continued scalability on the architectures of tomorrow.

Since it is not feasible to rewrite an entire production application from scratch for each new technology, it is important to understand how to introduce the desired portability and future-proofing in an incremental manner, and to do so without becoming tied to a proprietary language or programming interface. To this end, a case study detailing the porting of a production plasma physics code to accelerator architectures using the Open Computing

Language (OpenCL) [57], one of many available open standards for targeting accelerators [82, 83, 84], is presented.

In this Chapter we aim to develop an understanding of techniques which allow legacy Fortran codes (that are typically serial in nature), particularly Particle-in-Cell (PIC) plasma physics applications, to benefit from the increasing amounts of on-node parallelism present in emerging HPC architectures. The utility of OpenCL for future-proofing a production application, and the extent to which accelerator hardware can lead to application speedups in the PIC domain is presented.

6.1 Background

As previously discussed in Chapter 4, the core algorithm of EPOCH features a large amount of inherent parallelism: the calculations for the position and momentum of each particle are independent, and the electric and magnetic fields remain constant between loop iterations. However, the current accumulation step contains a write-conflict; any number of particles (which are close in space) may attempt to update the charge for the same grid point. This conflict prevents the loop from being parallelised in a naïve fashion (*e.g.* via the insertion of pragmas, or source-to-source translation), and necessitates some form of global synchronisation between work-items, in order to ensure that no two work-items can update the same cell simultaneously.

When EPOCH (and many similar codes) were originally developed, such factors did not need consideration – during single threaded scalar execution, the grid updates of multiple particles cannot conflict with one another. As hardware designs have become increasingly focused on parallel execution, inherently serial algorithms such as this will become more of a performance bottleneck. Indeed, the issues we encounter with EPOCH's current accumulation step are not specific to OpenCL, or to accelerators: the write-conflict complicates the use of parallelism on CPU and GPU architectures alike.

The extensible nature of EPOCH constrains the set of allowable algorithmic implementations, limiting the extent to which it is allowable to change the core algorithm, a consideration not held by other PIC applications. An example of this is that EPOCH's algorithm is required to ensure that the electric fields satisfy Poisson's equation, a condition that many non-relativistic codes do not have to meet. In this Chapter, we seek to determine whether these relativistic algorithms can benefit from the levels of parallelism present in accelerator architectures, or whether it will be necessary to adopt a fundamentally different approach.

Open standards such as OpenMP [83], OpenCL [57], and OpenACC [82] provide application developers a greater level of portability than third-party libraries and languages, since they are supported by multiple compilers and on multiple architectures. Such standards may also provide some level of future-proofing, with each updated specification introducing new constructs required to exploit recent architectural changes (*e.g.* OpenMP is adding support for the vectorisation of loops [59] and offloading computation to accelerators [84]). However, a common criticism of these standards is that although they guarantee *functional* portability, they make no guarantees of *performance* portability (*i.e.* the ability of a single source code to achieve good levels of performance on a wide variety of hardware). The development of performance-portable OpenCL codes has been the subject of previous work [28, 62]; including work in which OpenCL is utilised for the acceleration of wavefront [89] and molecular dynamics [88] codes.

The use of Graphics Processing Unit (GPU) architectures in the PIC domain has been explored in previous work. The importance of carefully partitioning the grid space has been discussed by Joseph et al., concluding that an efficient GPU PIC solution requires fine tuning and extended programmer effort [50]. They also discuss the importance of efficient use of shared memory on GPUs – this feature is not available on x86-based platforms (*e.g.* Central Processing Units (CPUs) and Intel Xeon Phi coprocessors), and may therefore negatively impact

the ability to offer portable performance. Stantchev et al. present a range of algorithms to perform grid interpolation, and discuss a range of potential issues including shared memory and thread contention [105]; whilst Burau et al. also show that PIC solutions have been shown to scale across multiple accelerators while maintaining numerical stability [17].

6.2 OpenCL Implementation

In the original EPOCH code base, and as shown in Figure 4.3 (Section 4.4), the `push_particles` subroutine consists of a deep loop nest spanning around 600 lines of code. This is reasonably typical of large legacy codes that have been optimised for serial execution, exhibiting several properties that map well to traditional CPU architectures – in EPOCH's case, particle data are stored in a linked list of structs, and are read/written only once in the entire loop, in order to improve cache performance. Such optimisations may actually be detrimental to performance on modern, parallel architectures: large kernels typically require many registers, and may cause register spilling; grouping dependent and independent operations in a single loop may completely prevent parallelisation (as explained in Section 6.1). Typically, array based data stores are better suited to SIMD execution. To address these concerns, the main kernel is *fissioned* into three code portions, corresponding to the algorithmic steps discussed previously in Section 4.4.1:

1. Moving the particles;
2. Updating particle momenta;
3. Updating the current.

The overhead of kernel fission in this case is relatively low. It requires only a small number of values to be recalculated between kernels, and allows the application of different optimisations to the three different kernels. It also separates

the update conflict from the majority of the compute, and the same fissioning process could therefore enable the use of auto-vectorisation/OpenMP on CPUs.

6.2.1 Particle Move

After loop fission, the particle move step is simple to represent in OpenCL. A single particle is assigned to each OpenCL work-item, and the OpenCL runtime is allowed to decide upon the best work-group size for a given device. Since there is no explicit use of shared memory, or other architecture-specific features, such a kernel design is expected to exhibit portable performance across various hardware types. The kernel is memory bound – it simply reads in particle positions and momenta, performs a small number of floating-point operations, and then writes out the new positions – and would therefore be expected that its performance will scale with memory bandwidth.

The only optimisations employed in this kernel concern the original particle data layout used by EPOCH. Whereas the original Fortran code uses a linked list of particle structs, the OpenCL implementation stores particle data in a Struct-of-Arrays (SoA) layout and thereby ensure that memory accesses to particle data are coalesced. Figure 6.1 shows an example of an OpenCL kernel, the structure of which is representative for all kernels used in this investigation, with an instance of the kernel being launched per work item.

6.2.2 Field Calculation

The second step of the particle push, in which particle momenta are updated based on the surrounding electric and magnetic fields, is also data-parallel following fission. However, unlike the particle move step, there are many optimisation opportunities. Each particle must read field information from the 27 cells surrounding its own (*i.e.* a 27-point stencil), and the optimisation of such kernels for GPU architectures is well-studied [73].

In the original Fortran implementation of EPOCH, particles are not stored in any particular order. Therefore, these stencil operations additionally become

```
1  __kernel void push_particle_inner_move_knl( ... )
2  {
3      for (size_t ipart = get_global_id(0); ipart < get_global_size
           (0); ipart += get_global_size(0))
4      {
5
6          const double ipart_mc = 1 / part_mc;
7
8          const double cmratio  = part_q * dtfac * ipart_mc;
9          const double ccmratio = c * cmratio;
10
11         const double part_weight = particle_array_weight[ipart];
12         const double fcx = idtyz * part_weight;
13         const double fcy = idtxz * part_weight;
14         const double fcz = idtxy * part_weight;
15
16         double art_x = particle_array_part_pos(ipart,1) -
               x_min_local;
17         double part_y = particle_array_part_pos(ipart,2) -
               y_min_local;
18         double part_z = particle_array_part_pos(ipart,3) -
               z_min_local;
19
20         const double part_ux = particle_array_part_p(ipart,1) *
               ipart_mc;
21         const double part_uy = particle_array_part_p(ipart,2) *
               ipart_mc;
22         const double part_uz = particle_array_part_p(ipart,3) *
               ipart_mc;
23
24         const double root = dtco2 / sqrt(part_ux*part_ux + part_uy*
               part_uy + part_uz*part_uz + 1.0);
25
26         part_x = part_x + part_ux * root;
27         part_y = part_y + part_uy * root;
28         part_z = part_z + part_uz * root;
29
30         particle_array_part_pos(ipart,1) = part_x;
31         particle_array_part_pos(ipart,2) = part_y;
32         particle_array_part_pos(ipart,3) = part_z;
33
34     }
35 }
```

Figure 6.1: OpenCL Kernel code example.

gather operations (*i.e.* uncoalesced loads); a collection of W particles (assigned to W work-items), may lie in W different cells and access $27 \times W$ memory locations. In order to improve this memory access pattern, domain-specific knowledge is exploited to reorder the particles before computing their momenta. Specifically, particles are sorted using a simple binning kernel, based upon a combination of their position and half-cell shifted position (to account for the Yee staggered grid described prevously in Section 4.4)

Although sorting in this way increases the probability that W particles will read from the same cells, it does not guarantee it; this prevents us from exploring other optimisations (such as using shared memory to cache cell data). In order to address this issue, three alternate levels of parallelism are considered: (i) assigning one work-item per particle (the default); (ii) assigning one work-group per cell; and (iii) assigning one work-group to some set of cells (henceforth referred to as a 'super-cell'). Each of these three levels of parallelism maps intuitively to different distributions of particles: (i) a very sparse distribution, with a small number of particles per cell; (ii) a very dense distribution, with many particles per cell; and (iii) a distribution somewhere in between, where the size of the super-cell is set so as to contain a specific number of particles. In cases (ii) and (iii), shared memory can be used to reduce the number of accesses to global memory.

6.2.3 Current Accumulation

In order to overcome the write-conflict present in the current accumulation step of the particle push, some form of global synchronisation between work-items must be introduced to ensure that no two work-items can update the same memory location simultaneously. There are a number of well-known approaches of this kind: critical sections; cell-wise (or super-cell-wise) locking; and atomic operations. However, implementing any of these approaches in a portable manner is challenging: different hardware has different levels of support for atomics (*e.g.* 32-bit, but not 64-bit integers); and all three depend on extensive use of

OpenCL's `atom_cmpxchg` routine, careless use of which can lead to deadlocks on some hardware (*e.g.* NVIDIA GPUs, where work-items are scheduled for execution in lock-step).

A further complication influencing the performance of atomic locks is the data access pattern. If two work-items attempt to update the same cell simultaneously, then one must wait until the other has finished – performance can therefore be increased by maximising the temporal distance between two accesses to the same memory location, in order to avoid contention between work-items. The end result is a performance trade-off between the field calculation and current accumulation kernels: the best case for one is the worst for the other. This is discussed further (alongside full performance results for the different approaches) in Section 6.3.

As an alternative to the current implementation, it is possible to replace EPOCH's current accumulation step with one more suitable for accelerator architectures. For example, one approach referred to as 'classic PIC' does not accumulate current densities at all, instead calculating the current density in a later step by summing over particles. However, this leads to the electric fields no longer satisfying Poisson's equation, and requires a corrector step which distorts the electromagnetic wave dispersion relation – this scheme is unsuitable for relativistic problems.

6.3 Results

The configuration of the experimental hardware is given in Tables 3.1 and 3.2. The reader's attention is drawn to the presence of two CPUs: the X5550 (which is based on the Nehalem microarchitecture) is used to compare the performance of the original Fortran code to that of the OpenCL implementation; and two E5-2670 sockets (based on the newer Sandy Bridge microarchitecture) are used as the baseline for comparisons between CPU and GPU architectures. For all code variants the Intel 13 compiler (with `-O3`) was used in conjunction with

Approach	X5550	C1060	C2050	K20
Crit. Section	0.003	–	0.260	0.855
Lock (Face)	0.001	6.118	0.054	0.088
Lock (Row)	0.001	0.396	0.016	0.011
Lock (Cell)	0.002	0.041	0.011	0.003
Atomic Add	0.002	0.026	0.008	0.002

Table 6.1: Comparison of execution times (in seconds) for different mutual exclusion approaches.

IMPI version 4.1.0.030. For the CPU based OpenCL runs, the AMD APP 2.8 Software Development Kit (SDK) was used, and the NVIDIA GPUs used the CUDA Toolkit 5.0. The OpenCL runtime was not available to us on the Sandy Bridge system (E5-2670) used later during CPU-GPU comparisons.

For the following experiments, EPOCH was used to simulate the interaction of two densely packed electron streams, with runtime options typical of normal conditions: each particle is assumed to have an individual mass; and a triangular pseudo-particle shape function. A modest problem size of 25 million particles (128 particles per cell, for a 58^3 cell problem) is used, with periodic boundary conditions. The CPU code was setup to issue manual prefetches between loop iterations (on account of particles being stored in a linked list), and loop fission is employed to facilitate a more accurate comparison between the three different kernel components. This fissioned version of the loop is ≈5–10% slower than the original Fortran code, due to the introduction of additional memory accesses. To avoid artificially inflating any claims about the performance of our accelerator-based solutions we include all known overheads (such as PCIe transfers) and acknowledge the importance of considering the impact of kernel-level speedups in the context of overall application time.

6.3.1 Effects and Portability of Optimisations

Mutual Exclusion

Table 6.1 compares the performance of three alternative mutual exclusion approaches (global critical sections, selective locking, and atomic adds), used to prevent write-conflicts within the current accumulation kernel. Execution times for multiple granularities of locking are reported, including: locking a face (*i.e.* using one lock per z co-ordinate); locking a row (*i.e.* using one lock per pair of y and z co-ordinates); and locking individual cells. All locks make use of a 32-bit `atomic_cmpxchg` operation.

There are several interesting trends in the data. On the CPU, we see that although using locks or atomic adds is faster than using a global critical section, the locking granularity has little effect on kernel performance. This is due to the low number of work-items (relative to GPU hardware) that are able to contend for any given lock. On the GPUs, performance improves as the mutual exclusion becomes finer, with atomic adds providing the best runtime in all cases; this is a somewhat surprising result, since it could be expected that the overhead of acquiring locks (via atomic operations) would be more expensive than the writes to memory. Also of interest is that NVIDIA's Kepler architecture is outperformed by the older Fermi architecture in the presence of coarse locks – this is likely to be a reflection of the greater levels of parallelism present in Kepler, and the increased lock contention this may cause.

To investigate the overhead of atomic adds on different architectures, the experiments were repeated using normal writes (allowing the kernel to write to conflicting memory addresses, and achieve the wrong answer). Although this baseline is unrealistic, it is used here merely to represent a "best case" for writing to global memory. The results (Figure 6.2) show that the overhead of atomics is inconsistent across different hardware: 2× on the CPU; 14× on the Tesla; 16× on the Fermi; and 2× on the Kepler. The performance of the atomics have been greatly improved with the introduction of new GPU architecture features, most

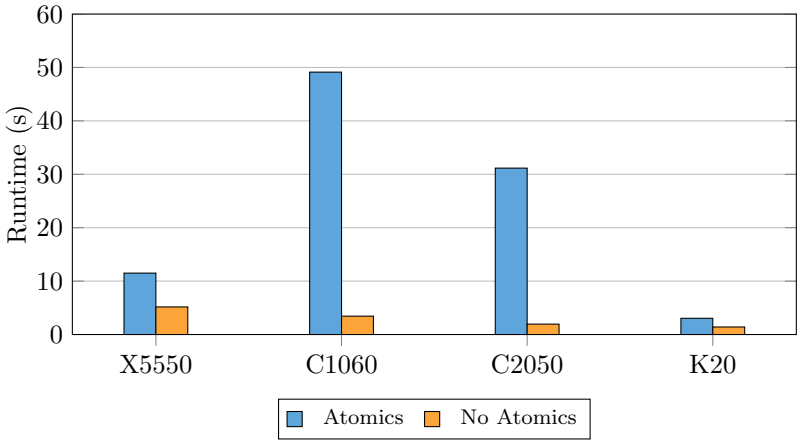


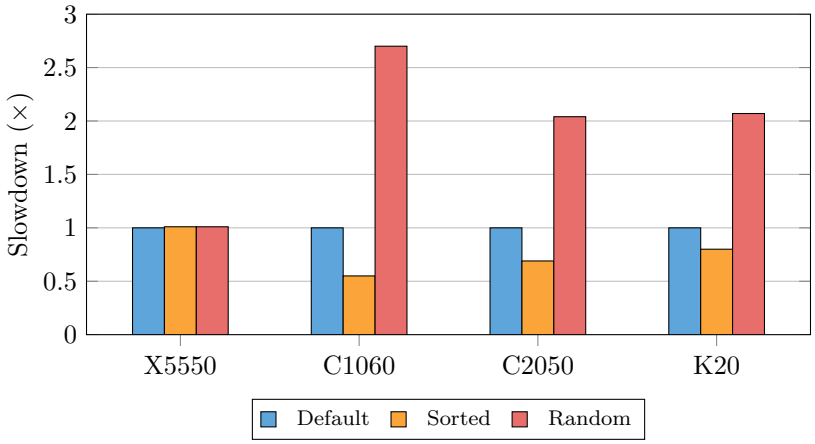
Figure 6.2: Overhead introduced by atomic operations.

prominently the Kepler architecture, but a 2× overhead is still undesirable for such an expensive kernel.

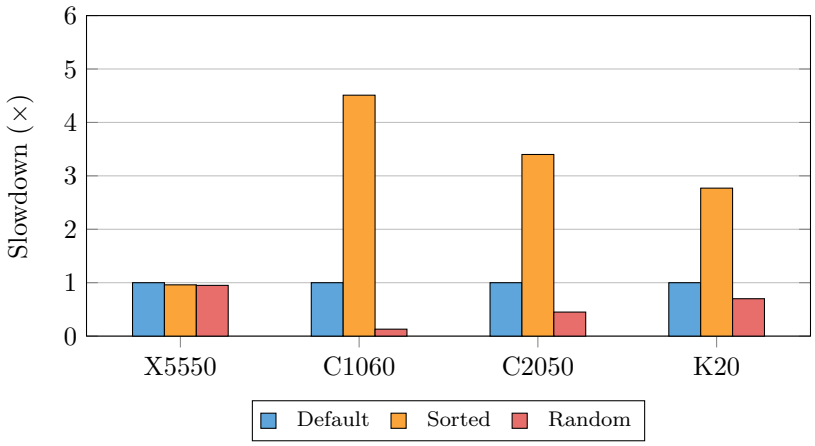
Since atomic adds provide the best performance on the architectures considered here, this version of the current accumulation kernel in used for all of the experiments that follow. However, it should be noted that this kernel has very weak guarantees on performance portability since, as noted previously, some OpenCL runtime platforms do not feature support for 64-bit atomics. This is an issue in software, rather than hardware. Due to software comparability it is not possible to run the OpenCL kernel on the X5550 processor using Intel's runtime, but it is possible using AMD's. Combined with the varying overhead of atomics on different architectures (and the inherent serialisation that will arise as parallelism, and hence lock contention, increases) there is clear motivation for further investigation of alternative algorithms in the future.

Particle Ordering

The graphs in Figure 6.3 present runtimes for (a) the field calculation kernel; and (b) the current accumulation kernel, using atomic adds. Three different particle orderings are compared: the default ordering used by EPOCH (Default); an ordering based on the particle's cell ID (Sorted); and a pseudo-random ordering



(a) Field calculation kernel.



(b) Current accumulation kernel.

Figure 6.3: Impact of sorting schemes on kernel runtimes.

(Random).

As discussed in Section 6.2, such orderings are expected to have a significant impact on the performance of both kernels, and this is reflected in the GPU results. On the CPU, however, we see very little effect. This is because the kernels do not make use of vector instructions, and thus the stencil operation is not a gather on this platform – the biggest bottleneck for the scalar implementation is cache performance. The size of the electric and magnetic field data is small (12 MB) in comparison to the size of the particle data (1.5 GB), but accessed

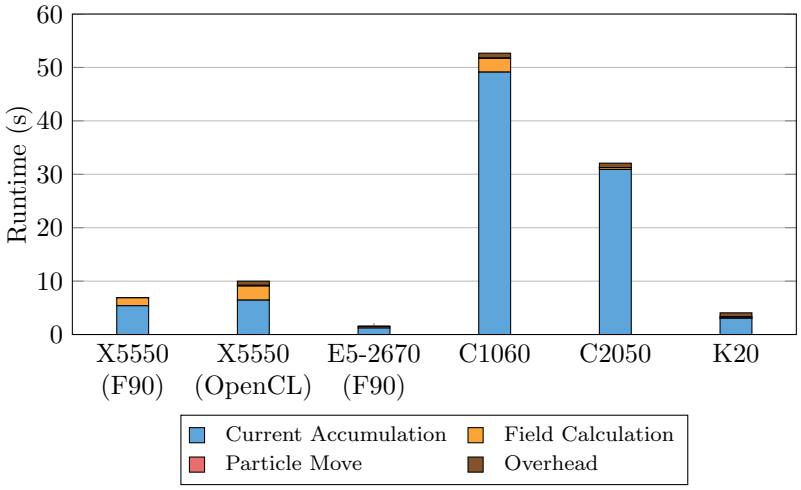


Figure 6.4: Best kernel performance across platforms.

more frequently, and therefore good cache behaviour is exhibited regardless of particle ordering. Also of note is that the impact of particle ordering decreases with each generation of the CUDA architecture – this may be attributed to the introduction and improvement of hardware caches.

Since current accumulation is so much more expensive than field accumulation, random particle ordering is used henceforth – taking a 2× performance penalty on the field accumulation has a smaller effect on overall runtime than a 4× performance penalty on the current accumulation.

6.3.2 Hardware Comparison

The graph in Figure 6.4 compares the runtime of the final kernel configuration (atomic adds, with a random particle ordering) on each platform. In addition to the performance of each individual kernel, any overheads (*e.g.* PCIe data transfers) are also listed, to facilitate a fairer comparison between architectures.

The performance of the OpenCL code on the X5550 almost matches that of the native Fortran code. The difference in performance is caused not by the current accumulation kernel (as might be expected) but by the field calculation kernel. This is due to the change in data layout from Array-of-Structs (AoS) to

SoA. This change spreads the data for a given particle over several cache lines. That the performance difference is so low is a positive result for the maintainers of EPOCH, suggesting OpenCL will provide a good route to explore accelerator performance without a significant negative impact on CPUs.

What is most clear from the results is that the current accumulation kernel is an issue, dominating the runtimes of all implementations on all hardware. The other kernels perform well, and in line with our expectations. However, the overhead is much larger than running the kernels themselves – hiding PCIe communication costs via pipelining, or making more kernels resident on the device, will clearly be a critical direction for future work following the acceleration of the current accumulation step. Something not considered here is how the performance of EPOCH on different hardware changes with the number of particles per cell. This impacts simulation accuracy, and current simulations are bounded by performance – running more particles per cell is desirable, but too expensive. It may be that GPU implementations only begin to offer greater performance for these larger, more complicated, simulations.

6.4 Summary

In this Chapter, the development of an OpenCL implementation of EPOCH, a production PIC code developed by the University of Warwick, is reported. The PIC algorithm is highly parallel, and should map well to accelerator architectures. However, as shown in this study, there are two issues that first need to be addressed in EPOCH: the highly serial nature of its legacy Fortran implementation; and the presence of write-conflicts in its current accumulation step.

Despite the promising initial results demonstrated in this work, the findings presented in this Chapter confirm that the current accumulation step (and in particular, its use of atomics) is the biggest barrier to both performance and portability – suggesting that a fundamental change to the algorithm is necessary

to fully utilise the massive levels of parallelism supported by emerging parallel architectures. The work in this Chapter directly motivates a use case for the development of a mini-app for EPOCH, which could directly build on the knowledge gained during this mini-benchmark investigation.

CHAPTER 7

Optimisation of Particle-in-Cell Simulations

Through the development of mini-applications, rapid investigation into known performance deficiencies can be performed. In this Chapter, the development and use of miniEPOCH is presented. The mini-app is specifically used to investigate a known time-step scaling issue within EPOCH and explore the following possible optimisations: (i) employing loop fission to increase levels of vectorisation; (ii) enforcing particle ordering to allow the exploitation of domain specific knowledge; and (iii) changing underlying data storage to improve memory locality. When applied to EPOCH, these improvements represent a 2.02× speedup in the core algorithm and a 1.55× speedup to the overall application runtime, when executed on EPCC's Cray XC30 ARCHER platform.

The continued development, maintenance and future-proofing of EPOCH represents a significant software engineering challenge. EPOCH is the result of decades of development by skilled domain experts – the code is feature rich, but equally large and complex. Code porting to explore the potential benefits of new compute architectures represents a significant undertaking, and the resulting benefits of this effort may indeed be small. *Mini-applications* – small code proxies that encapsulate important computational properties of their larger parent counterparts are often used to investigate these problems [6, 44]. The existence of mini-apps is built on the premise that, (i) although simulation codes may have millions of lines of source code, their performance is often dominated by a small subset of the code and, (ii) simulation codes may contain many physics models that are mathematically distinct, but in many cases exhibit similar performance characteristics. Mini-apps operate by encapsulating the most important computational operations and consolidating physics capabilities that have the same

performance profiles; they will typically be orders of magnitude smaller than their parent code, and as a result be easier to port, easier to improve, easier to extend, and less likely to be subject to restrictive licensing governing their use or distribution.

Section 4.4 provides background information about EPOCH, including a full description of the core PIC algorithm , which typically accounts for over 80% of the application runtime. The core algorithm represent a considerable computational workload and is currently expressed as a single code kernel in which the particle loop spans approximately 600 lines of source code. While particle-in-cell codes are well understood [11, 15, 17, 99], the application of the mini-app software engineering methodology to the field of plasma physics and PIC remains largely unexplored.

Previous work has been undertaken with the aim of providing flexible, concise environments for the development of PIC codes [21, 87]. GTC [68], at only 8,000 lines of code, is one such example of this; however, GTC is not associated with any parent code *per se*, and any findings associated with this code must still be translated to larger production codes in this code class through additional research. The research presented in this thesis is the first to develop and apply a PIC mini-app, which is associated with a large, production-code equivalent.

A known performance issue observed during the operation of EPOCH is that the duration of each time step increases as the simulation progresses. This problem is demonstrated in Figure 7.1, where a sharp increase in time-step duration can be seen until approximately 4,000 steps, after which the time-step duration stabilises. This observation is counter-intuitive, as there is no change to the kernel during runtime. This issue has persisted through many generations of EPOCH; in Section 7.2 we build on our knowledge and understanding gained in previous research [9] to address this time-step scaling problem, as well as using the new EPOCH mini-app to explore further code optimisation opportunities.

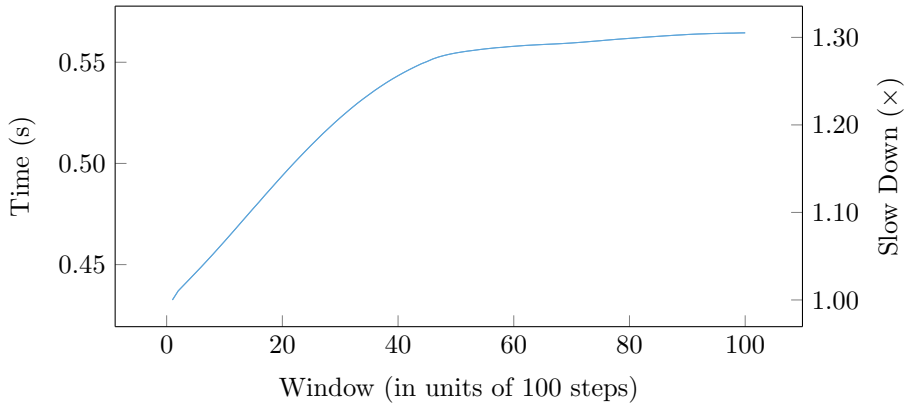


Figure 7.1: The duration of each time step in EPOCH as a simulation progresses. Each simulation window, of which there are 100 in total, contains 100 steps.

7.1 Implementation and Optimisation

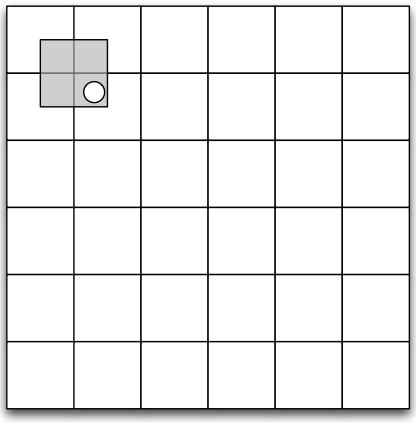
As discussed previously, the publicly available version of EPOCH uses a linked list to store its particles. While this itself does not present a problem, naïve implementations of linked lists offer no guarantees of contiguous memory access. This means that as data elements are inserted and deleted, memory allocations take place without consideration for locality of existing data, and considerable memory fragmentation can occur. This fragmentation can significantly impact performance, as modern hardware is optimised for contiguous memory loads, with each issued memory load fetching an entire cache line. By aligning data and promoting the grouping of data within cache lines, memory locality can be improved and one can reduce the effective bandwidth required to load the same amount of data from main memory. This effect can be seen when particles move across physical processor boundaries; as particles exit they are deleted and new particles added at arbitrary memory addresses. This means that although the initial particle allocation may be contiguous, the memory access pattern degrades over time, as a function of particle movement.

A second method for improving memory locality and, as a result, effective memory bandwidth, is to group data in memory such that it will cause subsequent loads to common addresses. In so doing, memory is more likely to be

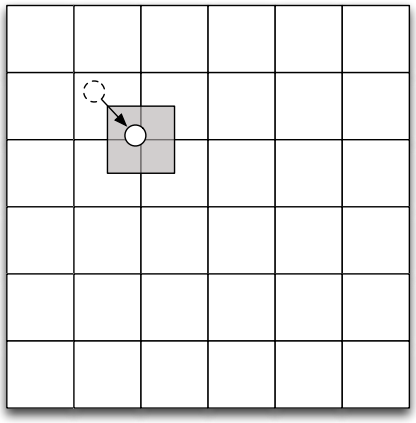
resident in cache when it is required and will reduce cache eviction and thrashing. To achieve this in miniEPOCH, we build on the idea of particle sorting first used in Chapter 6, and implement a particle sort to group spatially local particles in memory. Specifically, domain specific knowledge is exploited to group particles together in memory which will be accessed in the same way. Any set of particles which share a common grid-point mapping during the particle location update will share identical global memory access patterns, and have common intermediate values. By processing these particles simultaneously better cache re-use can be promoted, whilst simultaneously avoiding the need to recalculate redundant values. The grouping of particles is shown in Figure 7.2, with (a) representing the Yee grid rounding before the particle location move, and (b) representing it after.

A further performance limiting factor in EPOCH, is in its inefficient use of vector instructions. Typically, it is desirable to vectorise over the most computationally intense code regions in order to fully exploit Single Instruction, Multiple Data (SIMD) width. In EPOCH this means vectorising over the *particles loop*. However, in the current expression of the algorithm, such auto-vectorisation of particles is not possible due to a classical update dependency within the current deposition. This update dependency is key to the Particle-in-Cell (PIC) algorithm, so where the original code expresses the kernel as a single large loop, this is adapted in favour of expressing the code as three discrete steps. This is explored in the mini-app using loop fission, with pseudocode for this shown in Figure 7.3. As well as promoting vectorisation, splitting the code in this way also allows us to sort particles directly after they have been moved, giving us a stronger guarantee regarding particle reuse during the field-effect stencil. Hereafter these three components are referred to as the *move-*, *stencil-* and *current-kernels*.

While the sort itself does increase the amount of computation required for the execution, this is mitigated by two factors: (i) much of the computation for the sort can be done while the particle is in cache from the particle move; (ii)



(a) Pre-move



(b) Post-move

Figure 7.2: A diagram depicting the particle sort implementation in the context of the Yee grid rounding.

```
for all species do

    for all particles do
        ▷ Move particles.
        position ← position + momentum
    end for

    ▷ Optional Particle Sort.
    for all particles do
        ▷ Update momentum based on field effects.
        e_cell ← ⌊position⌋
        for all neighbours of e_cell do
            calculate electric field effects
        end for

        b_cell ← ⌊position + 0.5⌋
        for all neighbours of b_cell do
            calculate magnetic field effects
        end for

        momentum ← momentum + electric and magnetic field effects
    end for

    for all particles do
        ▷ Calculate and deposit currents.
        for all neighbour cells do
            calculate current
            deposit current
        end for
    end for

end for
```

Figure 7.3: Pseudocode depiction of EPOCH's modified PIC algorithm.

for particles that are grouped together, recomputing shared properties can be avoided, which was not previously possible. Further to this, it is also possible to employ the sort after the *stencil* kernel, allowing us to place guarantees on the order in which the *current* kernel updates global memory, which can in turn be exploited to remove the classical update dependency and achieve vectorisation. As auto-vectorisation of all three kernels is possible, vector efficiency can be further increased by performing scalar replacement on arrays where possible, and employing SIMD lane indexing to ensure all SIMD temporary arrays have coalesced data accesses.

7.1.1 Experimental Setup

Throughout this Chapter numerical results for the periodic interaction of two densely packed electron streams are reported. Each pseudoparticle is assumed to have an individual mass, and wide spanning third order b-spline stencil. Particle probes are disabled and, unless otherwise stated, a typical problem size of 128^2 grid-cells per core is used, with 32 particles per species, per cell, on a fully packed node. The results detailed in Section 7.2 were obtained from ARCHER, using the Intel 15.0 compiler, with the highest level of code optimisation enabled (`-O3`) with platform specific code generation (`-xHost`) enabled. PAPI 5.3.2.1 was used to gather the results of selected performance counters, including those used for recording cache misses and vector instruction counts.

7.2 Results

During the initial investigation of the increasing duration of time-steps in an EPOCH execution, it was believed that the primary contributor to the poor time-step scaling was increasing fragmentation of the linked list. As the particles move between MPI ranks, it was expected that the particle store would become more fragmented. As previously discussed, this was due to new particles being added to the store as they entered the domain, whilst others were removed as

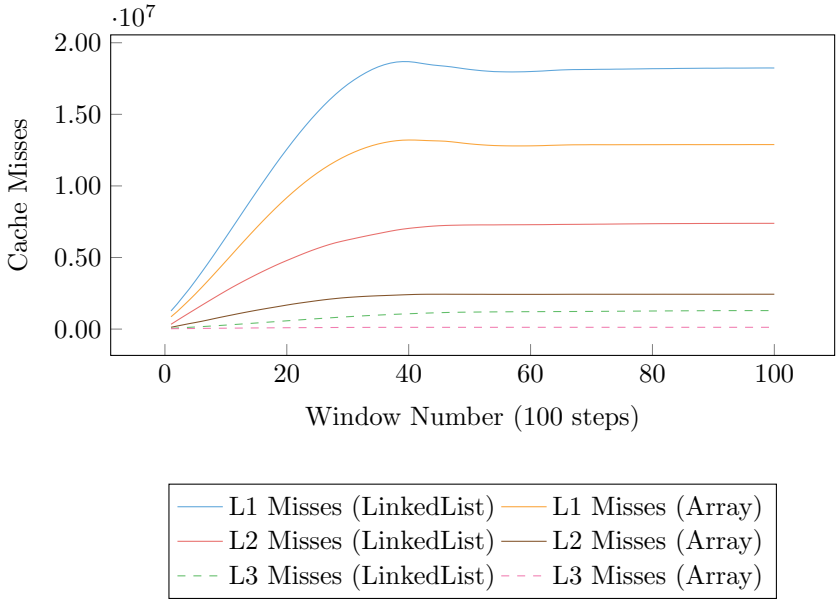


Figure 7.4: Cache misses during a simulation consisting of 100 simulation windows (each window containing 100 steps).

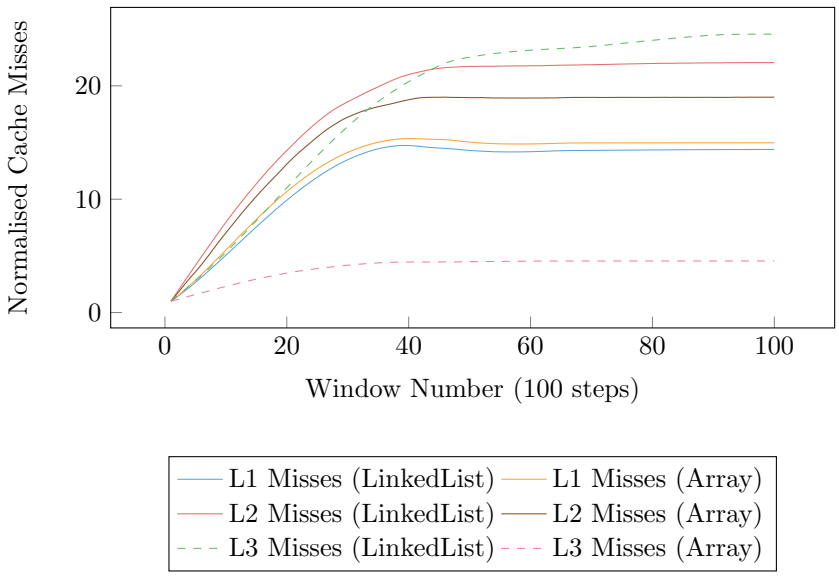


Figure 7.5: Normalised cache misses during a simulation consisting of 100 simulation windows (each window containing 100 steps).

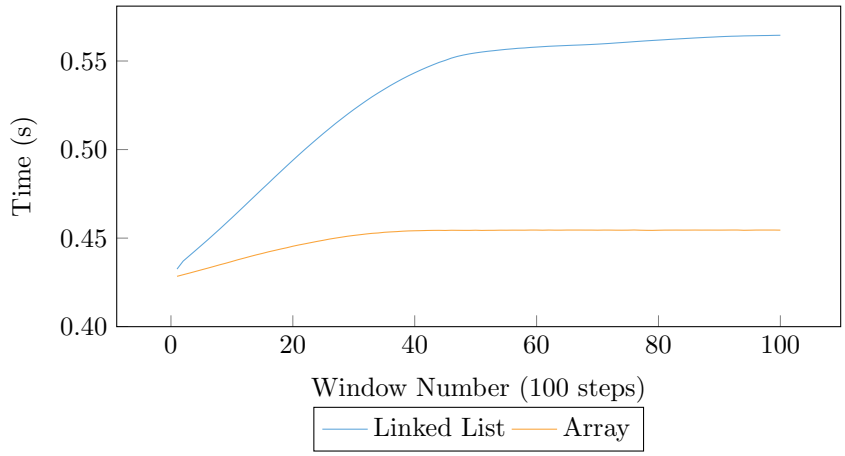


Figure 7.6: Time-step duration for a simulation consisting of 100 simulation windows (each window containing 100 steps).

they leave. Figure 7.4 shows the typical cache miss rates for an unsorted, linked-list implementation of EPOCH, while Figure 7.5 shows this data expressed as relative differences. We can clearly see that as the simulation progresses, the cache-miss-rate increase is strongly correlated with overall runtime, and peaks at approximately 4,000 steps. After 4,000 steps, the cache miss rates increase at a much reduced rate. While this data strongly suggests that the increase in runtime is caused by the change in cache miss rates, it provides no clue as to its origin.

To investigate this further, the mini-app was used to implement an alternative, array-based, particle store. This allows for contiguous access to particles, avoiding any fragmentation. This greatly improves the time-step scaling of the mini-app, with Figure 7.4 showing the representative changes in cache misses for a version of EPOCH with these changes applied. It is clear that this change in implementation only partially addresses the problem, with cache hit rates still scaling poorly as time-steps progress. The readers attention is however drawn to the marked decrease in L3 cache misses, which explains the substantial benefit to runtime scaling seen in Figure 7.6 for the array version of EPOCH. Whilst it is clear from Figure 7.6 that an array version of EPOCH benefits the overall runtime, the cache miss figures in Figure 7.4 identify that a secondary prob-

lem still exists. Further analysis of the cache miss rates directed attention to the large stencil required when applying electromagnetic effects to the particle momenta. Increasing disorder in the particle list may explain the poor cache and memory behaviour, it would increase the probability of cache pressure, and the probability of spilling during this stencil-gather. The strategy here was to periodically sort the particle store, and in so doing investigate the effect particle disorder had on both cache misses and runtime. Such a sort remedied the problem, and allowed for near perfect time-step scaling with a maximum deviation of 0.03% over ten thousand time-steps.

7.2.1 Vectorisation

Having arrived at a significantly improved array-based version of miniEPOCH, efforts could then be focused on optimising EPOCH to ensure that it was better able to utilise current and future hardware. At the outset of this study, EPOCH was unable to exploit vector operations in its main kernel. The reason for this was a classic update dependency when accumulating currents, with multiple particles possibly having to write to the same array location concurrently. Given the large size of the initial kernel (approximately 600 lines of code), loop fission could be used to great effect in order to separate out much of the non-dependant computation which was SIMD-parallel safe. The previously mentioned sort, combined with iterating over particles on a per vertex basis, allows us to hoist loop invariant calculations so that they could be performed once per vertex. This offered a decrease in the overall required computation, and allowed the compiler to better predict memory access patterns into global memory.

Through code modifications, which were first tested in the mini-app, we were able to ascertain that the restructured fissioned kernel was able to successfully exploit vector instructions. Figure 7.7 shows the SIMD scaling of kernel runtime for 256-bit AVX (4 doubles), 128-bit AVX (2 doubles), and for the code operating without vectorisation. Reasonable SIMD scaling is seen for all kernels, except for *move*. This is because of its memory-stream-like structure, and

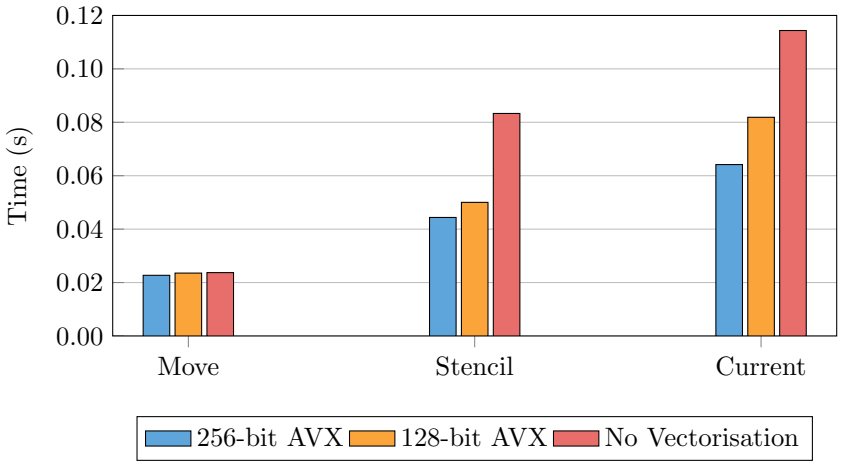


Figure 7.7: SIMD scaling of miniEPOCH AoS kernels for 256-bit AVX, 128-bit AVX and for the code operating without vectorisation.

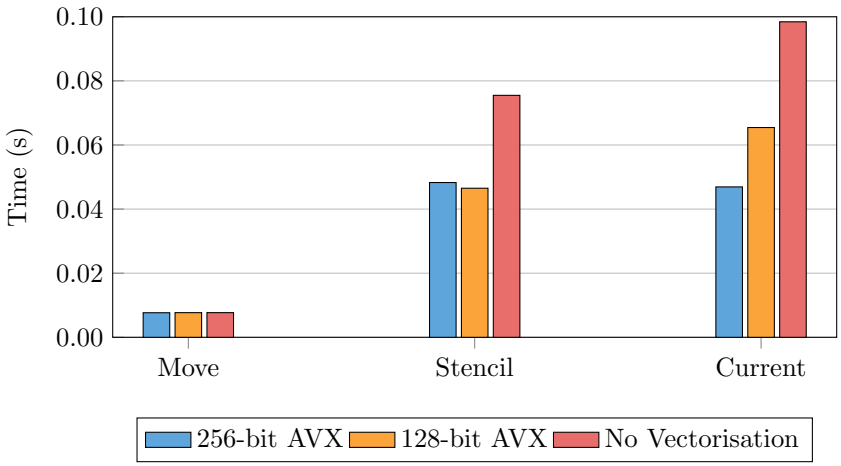


Figure 7.8: SIMD scaling of miniEPOCH SoA kernels for 256-bit AVX, 128-bit AVX and for the code operating without vectorisation.

the lack of Floating-Point Operations per Second (FLOP/s) to hide the memory accesses. The *stencil* and *current* kernels represent the majority of the time spent in miniEPOCH, and demonstrate significantly improved performance with SIMD width. Such improvements will yield further gains if the current trend of increasing SIMD width continues.

7.2.2 Memory Layout

Code performance can be improved further through the consideration and implementation of alternate memory layouts used for the particle-array storage. Not only does this change how data is accessed, it also determines the number of concurrent memory streams the processor has to track during pre-fetching. Typically, particles are stored in an Array-of-Structs (AoS). By storing particles as an AoS, a single memory stream is required, with each particle loaded bringing with it all particle properties from main memory. This effect holds regardless of the number of properties used in the given kernel, and can represent a significant overhead for kernels which require fewer fields. When processing multiple array elements, as is typical in SIMD, loads must be gathered from memory, and any writes scattered, incurring a performance cost and increasing the latency of the memory operations. Alternative approaches include a Struct-of-Arrays (SoA) and a more complex hybrid, Array-of-Structs-of-Arrays (AoSoA) (which aims to combine the benefits of both SoA and AoS). A brief overview of these memory layouts is found in Figure 4.1 (Section 4.2), where in our experiments SIMD width was typically 4, as is typical when operating on doubles with 256-bit AVX.

For the SoA data layout, single particle properties for multiple particles are stored together in an array. This means that under SIMD operation, single properties from multiple particles can be loaded in one contiguous and aligned load, at the expense of tracking a different memory stream per property required. This eliminates any potential for gather/scatters, and is often favourable when only a few particle properties are required. With AoSoA, groups of N elements

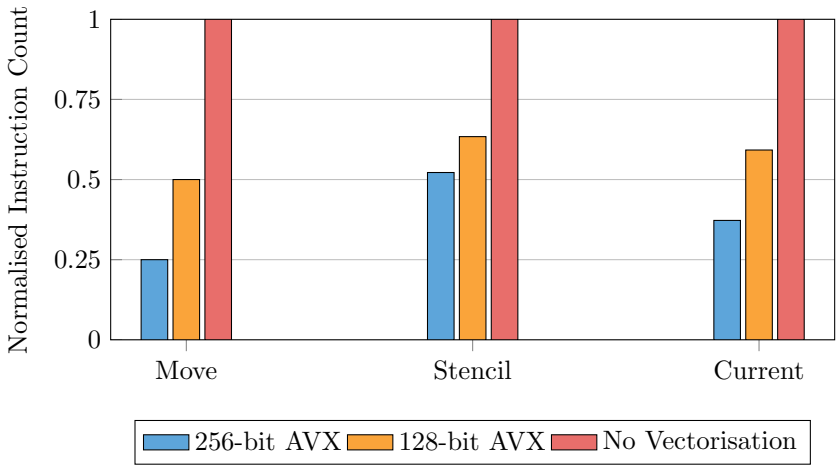


Figure 7.9: Normalised instruction counts per kernel for different SIMD widths for the SoA memory layout.

of each property are stored together, in order, where N is typically a function of vector length. This approach attempts to combine the benefits of both SoA and AoS, but comes at the expense of vastly increased complexity and an indexing overhead.

In the current kernel configuration, a particle has 6 properties and stores 1 intermediary value (all types are `doubles`). The *move* kernel requires 5 such properties, the *stencil* kernel requires 6 and the *current* kernel requires 4. To investigate enhanced SIMD scaling, the mini-app was ported to each of the alternative memory layouts – this provides an excellent example of where mini-apps allow rapid code exploration, which might not otherwise be possible on full production codes. Figure 7.8 shows the vector scaling of the SoA implementation, which compared to Figure 7.7 shows that the performance is favourable in the kernels requiring fewer particle field accesses, and generally favourable overall. This result is largely due to the more effective use of data loaded using SoA, as no bandwidth is wasted. Figure 7.9 shows the relative difference in instruction counts for varying SIMD widths, as recorded by PAPI. For good vector scaling one would expect to see the number of instructions executed decrease as a function of SIMD width. Both the `move` and `current` kernels scale as expected, but

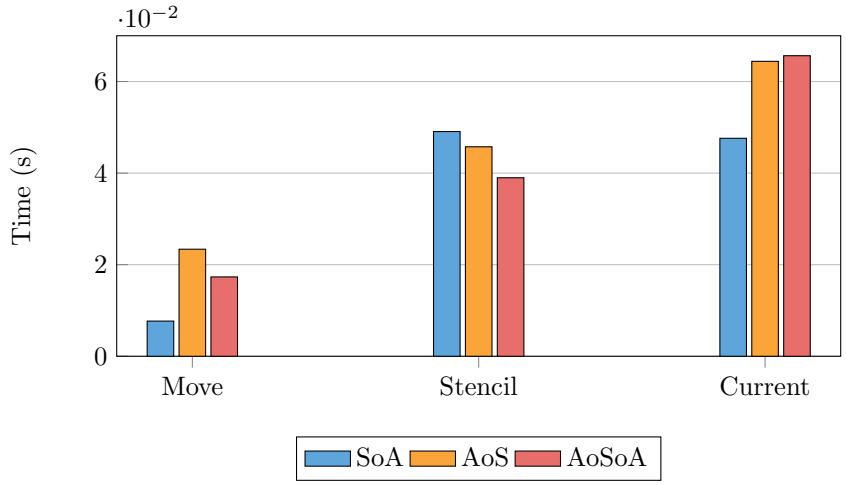


Figure 7.10: Kernel runtime for different data layouts.

the `stencil` kernel is only able to partially benefit from the increase in SIMD width.

Figure 7.10 shows the overall performance as a result of each memory layout. SoA performs better as fewer particle properties are required, but as more particle properties are required, AoS outperforms SoA. AoSoA pays the cost of masked hardware instructions due to the sparse particle grouping used, a cost which will be much reduced in future hardware generations and has already been much reduced on the Intel Xeon Phi product range. Each kernel benefits differently from the change in array layout, largely due to cache pressure and required memory bandwidth.

7.2.3 Parent Code Optimisation

As is typical of mini-app optimisation studies, the goal of the investigation was to map the improvements back to the parent code in order to facilitate improved scientific investigation. Figure 7.11 shows the overall runtime for an optimised version of EPOCH, as well as a comparison against the original EPOCH code base and an array based implementation. The array version shows a small increase in performance over the original implementation due to the lack of memory fragmentation and increased memory locality. The optimised version

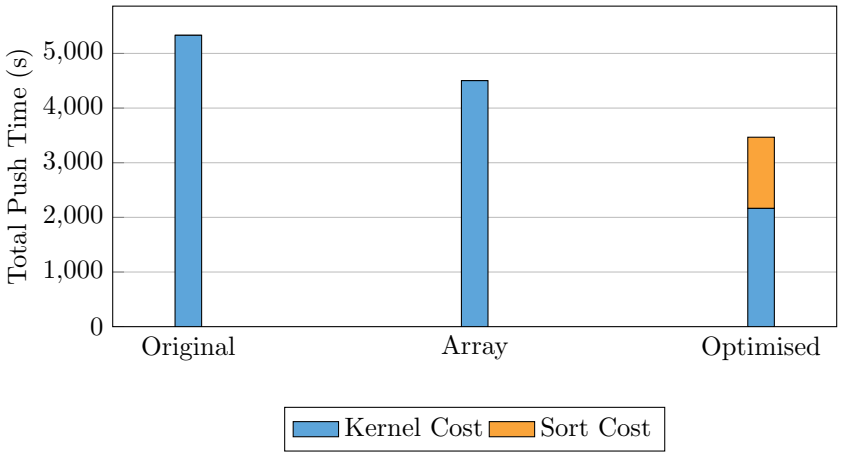


Figure 7.11: Overall runtime for the original and optimised versions of EPOCH for a 10,000 step run. The additional *sort* cost is highlighted for clarity.

of EPOCH represent a considerable improvement in code performance. It includes all previously discussed improvements, including spatial sort, increased vectorisation, loop-invariant code hoisting, array scalarisation and, coalesced temporary SIMD arrays. These optimisations deliver a notable improvement to production code, and successfully demonstrate the value of a mini-app-based investigation. These improvements, recorded on ARCHER demonstrate a 2.02× speedup in the core EPOCH algorithm and a 1.55× speedup to the overall application runtime.

7.2.4 Particle-Per-Cell Scaling

As part of the future proofing of EPOCH, a portion of this work looks to improve its Particle-Per-Cell (PPC) scaling. In PIC simulation, the accuracy of the result is most strongly governed by the number of particles in the simulation, typically described as the number of PPC. By increasing this number, pseudo-particles can represent fewer real world particles, and the gap between simulated and real world particle counts decreases. This allows for more complex interactions, as more bodies are involved. As the number of PPC increases, the amount of work also increases, typically linearly. In this Section we discuss a technique to enhance the way in which EPOCH can deal with these increasing workloads

needed for successful future operation, and provide a performance analysis in the context of miniEPOCH. To do this, we build directly on top of the previous sorting work which exploits the spatial ordering of the particles, and implement a specialised kernel for regions of particles which do not cross grid boundaries during the particle push. In so doing, the particles only need to be sorted once after the initial half timestep update in the *move* kernel, with the sort cost not needing to be paid again after the *stencil* kernel as was the case previously.

The specialised kernel can then be written knowing the particles do not cross any grid boundaries, meaning some simplifying assumptions can be made. In the default kernel, two arrays of coefficients need to be matched based on the movement of the particle. In a kernel with the assumption of no particle movement, this matching is entirely deterministic. This can be leveraged to decrease both the memory and compute needed to perform this, reducing the stencil from 7×7 to 5×5. Whilst this may only seem like a modest reduction, it decreases the amount work by near 30% in each direction. Furthermore this stencil is iterated across fully in a tight two dimensional loop, representing a reduction in work by nearly 50%. A range of additional benefits to this technique also exist, including but not limited to: reduced memory footprint; reduced data initialisation; lower effective memory bandwidth; and, better exposure of information to the tool-chain during compilation. Finally, by establishing a tighter bound on the physical domain each particle can interact with (implemented as shared global memory), this can give more control and flexibility when multi-threading, as it increases the maximum amount of concurrency without overlap.

By inspecting Figure 7.12, it can be seen that the PPC scaling of the original EPOCH code is perfectly linear, with an increase in PPC work being mirrored exactly in runtime. This can be directly contrasted with the optimised version of the mini-app, which when comparing 32 PPC to 512 PPC takes only 9.91× as long to do a 16-fold increase in work. This scaling represents a time increase between 1.61× and 1.87× compared to the expected 2×, with an average of 1.78× over the sampled data. This super-linear scaling indicates that increased

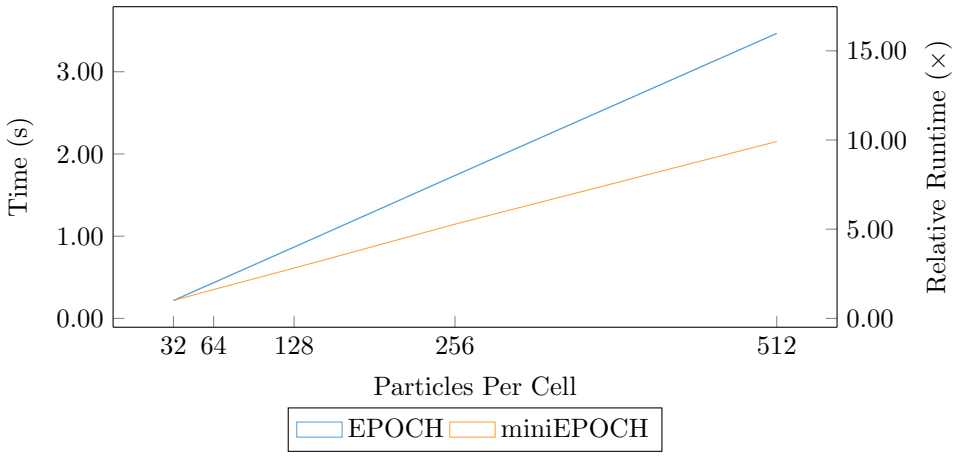


Figure 7.12: The PPC time scaling of the original EPOCH code base and the optimised mini-app.

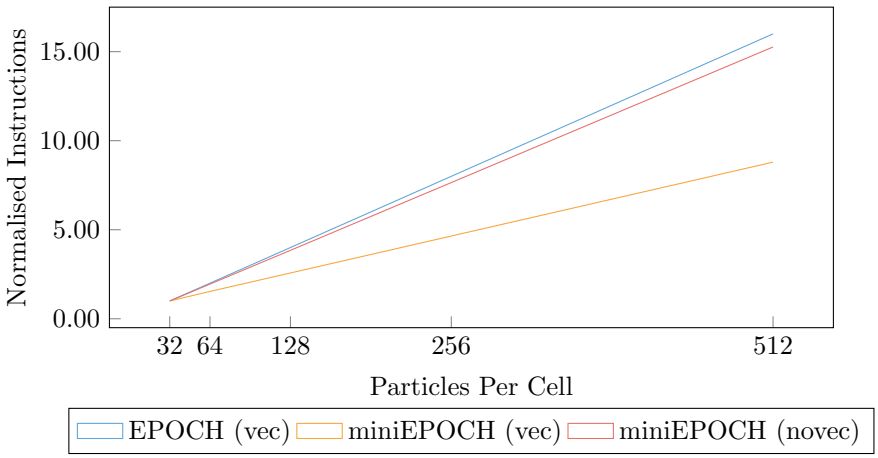


Figure 7.13: The PPC instruction count scaling of the original EPOCH code base, and the optimised mini-app.

workloads are not only possible, but in fact favourable.

The root cause of these improvements is identified in Figure 7.13, which analyses the executed instruction counts for both EPOCH and miniEPOCH. We can see that with vectorisation disabled for miniEPOCH, the instruction throughput of both codes is very similar. However with vectorisation of miniEPOCH enabled, we can see that as the number of particles per cell increases, the vectorisation efficiency increases, leading to super-linear scaling in the number of instructions executed relative to the amount of work to be done.

7.3 Summary

Despite recent successes at the large laser-based Inertial Confinement Fusion (ICF) device at the National Ignition Facility at Lawrence Livermore National Laboratory (LLNL), the community remains some distance from being able to create controlled, self-sustaining fusion reactions. ICF represents one leading design for the generation of energy by nuclear fusion and computing simulations supporting ICF continue on some of the world's most powerful supercomputers. The research presented in this Chapter focuses on EPOCH, a fully relativistic PIC plasma physics code, developed by a leading network of over 30 UK researchers. A significant challenge in developing large codes like EPOCH is maintaining effective scientific delivery on successive generations of high-performance computing architecture. To support this process, the use of mini-applications was adopted – small code proxies that encapsulate important computational properties of their larger parent counterparts.

Through the development of miniEPOCH, we investigate known time-step scaling issues within EPOCH and explore possible optimisations. Not only have we developed a novel mini-app for the parent code EPOCH, we all address two key issues of: (i) the increasing time-step duration during simulation runtime and, (ii) high levels of cache miss rates due to particle-store fragmentation. Overall, these improvements demonstrate a 2.02× improvement to the

core EPOCH algorithm and a 1.55× speedup to the application runtime.

In future work it is hoped that the performance impact of the additional particle sort, highlighted in Figure 7.11, can be reduced. Whilst the current implementation of the sort remains largely unoptimised, an algorithmic change would likely yield a considerable improvement. The current implementation performs a full particle sort despite large portions of the data remaining sorted. A sorting algorithm which exploits this feature could greatly improve overall application performance. Additionally, a trade off between sort-cost and the performance improvement could be achieved, by tracking sorted data regions and sorting periodically, rather than requiring a fully ordered sort every timestep. Unfortunately such investigation fell outside of the scope of this work and is this reserved for future work

Finally, as part of this work an OpenMP port of EPOCH has also been developed. The results of this port demonstrate good on-node scaling, and future work will build on this code-base to develop an Intel Xeon Phi code version. This version of miniEPOCH would be able to utilise the increased SIMD width offered by the Intel Xeon Phi, and could facilitate a study assessing the viability of heterogeneous and accelerated hardware platforms for PIC codes.

CHAPTER 8

Discussion and Conclusions

The work presented in this thesis highlights the importance of understanding code performance as a key component of a successful optimisation effort. It highlights the importance of Inertial Confinement Fusion (ICF) research, and demonstrates the possibility of improving ICF code performance on both current and future architectures. Here this is expressed through the development of performance models, mini-apps, and portable code variants; each guided by the advice of domain experts. Whilst the work in this thesis focuses on ICF simulation the message it carries can be applied to many areas of High Performance Computing (HPC), only becoming more pertinent as code development initiatives are accepted as a requirement to prepare current simulations for future compute platforms.

Chapter 5 shows the power of performance modelling to achieve high levels of code understanding. A novel performance model for the code Lare is developed, and the model is shown to provide performance predictions for optimised code on both future and current hardware. Such predictions can guide procurement, help target optimisation efforts and allow users to better understand code performance. This trifecta of benefits is only set to become more important, as compute hardware begins to diversify, and heterogeneous hardware becomes commonplace.

Chapter 6 documents the process of developing portable code which is able to utilise both current and future hardware. A novel Open Computing Language (OpenCL) code variant of EPOCH is developed, which is the first known version of EPOCH able to successfully make use of accelerated architectures. We provide a performance comparison, and offer our thoughts on how the code

may need to change in order to effectively utilise emerging hardware.

Finally, Chapter 7 builds on the previously presented ideas, and shows how mini-apps can be powerful tools to investigate code performance on new platforms. We show how our mini-app version of EPOCH (miniEPOCH) can be used to guide future code development, as well as efforts in performance tuning. Our optimisation study readily reveals a variety of unexposed improvements to EPOCH, which when mapped to the main code deliver a performance increase of over 2×.

8.1 Limitations

The primary limitation of this thesis, is that the discussed techniques focuses specifically on the applications Lare and EPOCH, and not on a more diverse range of applications. Although this may initially seem to limit the generality of the optimisations of programming techniques presented, these codes were chosen because they are representative of their respective code classes which play an important role in the area of ICF simulation.

A further limitation of this work is that the work discussing EPOCH is limited to a sub-set of runtime parameters and physical simulations. Most notably, the work only considers collisionless Particle-in-Cell (PIC). EPOCH can be parameterised to run with the inclusion of a collision algorithm which is based on the approach for collisional PIC by Sentoku and Kemp [103], however, the majority of real world workloads do not need to make use of this. At high temperatures and low densities collisional effects in plasmas are generally considered minimal, and therefor have a negligible impact on the simulation result. Whilst enabling collisions may marginally increase the accuracy of the answer, it would come at the cost of increased runtime. Although this work was carried out without consideration for collisions, it is believed that some of the proposed codes change may in fact provide benefits to the existing algorithmic implementation. The collisions may benefit from the spatial memory-locality

afforded by the sorting schemes, and could be extended to explicitly leverage the spatial ordering of particles. It is unlikely, however, that the current collision scheme would map well to heterogeneous architectures due to its serial nature, but a body of previous work shows particle collisions and interactions achieve good performance on heterogeneous architectures [37, 91].

Finally, the lack of objective metrics used for evaluating code performance limits the applicability of this work. Much of the work presented focuses not only on performance, but also on the important ideas of portability and future-proofing code performance. Both of these ideas are underrepresented in the metrics provided through this thesis, as runtime is often the primary focus. This is especially true of Chapter 6 in which a portable, hardware agnostic version of EPOCH was developed. If such a version was adopted into the main code base, the effort of future porting efforts could be reduced or entirely removed, with only performance tuning required. It is this idea that despite being hard to quantify, the introduction of the mini-app in Chapter 7 addresses by providing a platform for decreasing programmer effort and promoting productivity.

8.2 Implications

Due to the legacy nature of many scientific simulations, much of the discussion so far has been limited to the maintenance of existing code bases. Code developers are now beginning to recognise that the changes to hardware designs are sufficiently significant that the re-engineering of key scientific simulations may be required. A significant implication of the research presented in this thesis is that the lessons learnt and the outlined findings can also be applied to this re-engineering process, or to the development of new simulations. These real world lessons learnt can be combined with well established theory, such as the Architecture Tradeoff Analysis Method (ATAM) [55] to help inform the process and to ensure new HPC simulations are able to efficiently leverage modern architectural features.

ATAM is a software evaluation method which focuses on assessing architecture suitability. It does this by identifying performance trade-offs and key sensitivity points. Often this includes evaluating the effect a given architecture has on a range of attributes and scenarios which can occur within the software execution. For ICF codes, these scenarios will include many aspects of the research included in this thesis, including but not limited to: particle ordering; particle storage design; data access patterns affecting vectorisation; load imbalance; and data re-use.

A technique such as ATAM, and the research presented in this thesis will clearly identify the following issues as key factors to consider during the development of a new ICF, PIC, and Magnetohydrodynamics (MHD) codes:

Data Ordering

Data ordering, specifically particle ordering in the context of PIC codes, should be of the utmost concern and consideration during the development of new scientific simulations. The cost of poor data ordering was hidden on historical hardware generations by other overheads, but is now key to performant code operation. This is true for all current generations of hardware, and is paramount to exploiting many hardware specific features such as efficient Single Instruction, Multiple Data (SIMD) utilisation, and effective use of the cache hierarchy. Scientific simulations can no longer afford to be developed without this being a primary concern.

Flexible Code Design

When many of the legacy simulation codes were developed, they were tightly coupled with the hardware on which they were designed. This approach is no longer viable as hardware designs becomes more diverse the life-span of software expands. Instead efforts need to be made to try safe guard portable performance, as well as efforts to facilitate on-going code maintenance and modernisation. This can be done through abstraction and software engineering, or through the

use of programming models such as COMPSs or Charm++. The work presented here highlights the importance of portability, and also the benefits of ensuring a code's design is not tightly coupled with a specific architecture.

The discussed techniques are applicable to all comparable codes. The work in Chapter 6 demonstrates how ICF codes which iterate over a particle list can be generalised to run on accelerator platforms, whilst the work in Chapter 7 shows a range of optimisations which can be applied to such codes.

8.3 Future Work

The work presented in this thesis is amenable to a variety of extensions, and further work. In this Section we present an overview of how the work is currently being taken forward, as well as provide a discussion of an Many-Core version of miniEPOCH which has been developed to target the Intel Xeon Phi product range. As part of the work in this thesis, miniEPOCH will be made publicly available to the wider scientific community.

8.3.1 Many-Core Investigation

As part of the work ensuring portable future performance, an OpenMP4 variant of miniEPOCH has been developed to target many-core architectures such as Intel Xeon Phi. Unlike many efforts porting code to Intel Xeon Phi, this work does not use the supported offload capability, instead favoring resident execution. To facilitate this, the code version extends the most heavily optimised version of miniEPOCH presented in Chapter 7, and employs OpenMP pragmas to expose additional parallelism for threaded execution. These pragmas are added such that particles in each sorted bin can be executed in parallel, with the aim of fully saturating all 240 threads of execution. In order to achieve maximum performance on the Intel Xeon Phi product range, the phyiscal cores need to be oversubscribed with at least 2 threads per core (with 4 being typical). This is due to the hardware's inability to issue vector instructions every cycle,

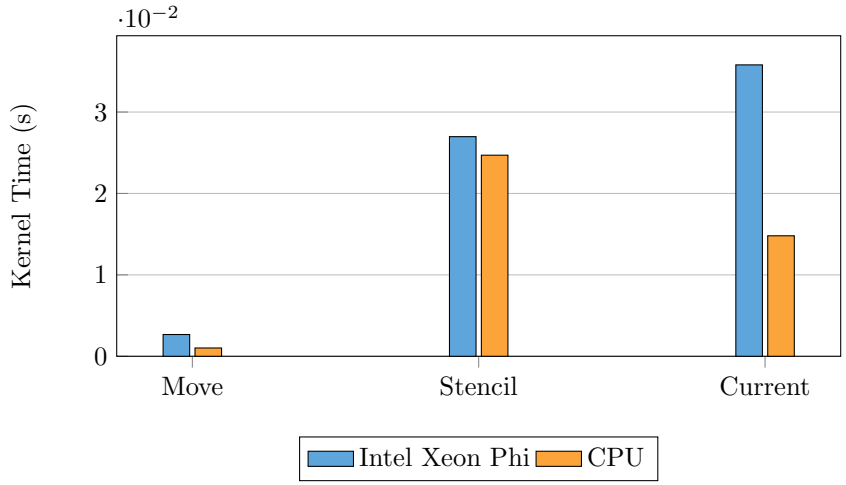


Figure 8.1: A comparison of Intel Xeon Phi and CPU kernel performance figures.

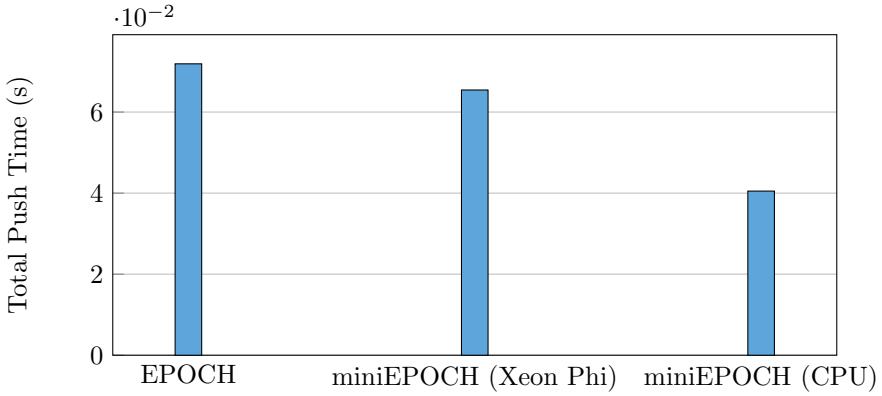


Figure 8.2: Push time comparison for the optimised versions of miniEPOCH and EPOCH.

instead issuing them on every second clock cycle.

Figure 8.1 compares the per kernel runtime of the CPU and Intel Xeon Phi variants of EPOCH. We can see that near comparable results are achieved for the *move* and *stencil* kernels. The *current* kernel scales less favorably, and this issue is still under investigation. Ensuring a fair comparison between different hardware platforms is non trivial, as a variety of metrics can be used. As previously discussed, Floating-Point Operations per Second (FLOP/s) can be a very misleading metric, as it only states a theoretical maximum – a number which may be very hard to achieve. Other candidate metrics include a com-

parison of Thermal Design Power (TDP) to assess running costs and thermal envelopes; and cost for a comparison of economic viability. For the Intel Xeon Phi card used, and the E5-2697v2 used the total FLOP/s counts are 1036.8 and 1208.3 respectively. Similarly, the TDP values are 260W and 300W respectively. The similarity between these numbers further supports the idea that the hardware used can offer comparable performance. It should also be noted that TDP figures for the Intel Xeon Phi includes additional overheads not required by stand alone Central Processing Units (CPUs). Such overheads include the additional power required by the card infrastructure, as well as powering the on-board memory.

Finally, Figure 8.2 shows the total push time for each code variant. Reasonable performance is attained on the Intel Xeon Phi, however it still underperforms when compared to the traditional CPU architectures. Such a result does however suggest that PIC codes may be able to successfully make use of future generations of accelerated hardware. As software support for OpenMP4 increases, other hardware will become targetable using the same source code, including Graphics Processing Unit (GPU) based architectures. This represents a significant step towards portable performance on future architectures.

8.3.2 Single-Precision EPOCH Code Variant

As part of an ongoing investigation, a single precision code variant of EPOCH is under investigation. Initially, miniEPOCH will be used to assess the viability of the technique, and if successful, changes will be mapped back to EPOCH. This single precision code variant focuses on promoting the use of the single precision data type (32-bit `floats`), in place of double precision numbers (64-bit `doubles`) throughout the main kernel. Single precision data, however, is inherently less precise that its double precision counter part, as it uses fewer bits to represent the same numeric value. Typically this will mean a trade off between speed of computation and simulation accuracy.

Techniques to bound the loss of accuracy do exists however. In order to

retain the levels of accuracy needed, and to limit the effect of numerical errors, a domain-specific field cleaner known as a Marder pass will be added [70]. This will be represented as an additional step in the main EPOCH algorithm, undertaken after the particle push has completed. As the required number of bits switching from double to single precision reduces the required bandwidth by a factor of two, and enables SIMD units to process twice the amount of data per instruction. Such changes could represent a significant step towards improving EPOCH's performance, and helping the code base to exploit increasing SIMD width and other modern hardware features. To regain the lost accuracy, the Marder pass calculates the difference in charge-density, and adjusts the simulation using the presence of a 'pseudo-current'. This technique has previously been shown to be successful in the PIC code VPIC [15] despite this additional overhead, so will likely also provide a performance benefit to EPOCH. This project, however, is still currently under development with no initial results being available. The core algorithm has been converted to use single precision, but costly data conversion operations are still present.

8.4 Final Remarks

This thesis represents a brief window into the diverse and fast-paced world of HPC research. With the rise of exascale computing, HPC will be faced with many upcoming issues which promise to be as exciting as they are challenging. The work presented in this thesis touches on many of these key issues, with a particular focus on those most pertinent to exascale ICF research, but sadly leaves many more unaddressed. The key themes of this thesis – performance portability, heterogeneous architecture, code optimisation, algorithmic changes – will be at the forefront of HPC research for many years to come.

Bibliography

[1] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: incorporating long messages into the LogP model - One step closer towards a realistic model for parallel computation. In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, SPAA '95, pages 95–105, New York, NY, USA, 1995. ACM.

[2] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, pages 483–485. ACM, 1967.

[3] K. Antypas. NERSC-6 Workload Analysis and Benchmark Selection Process. *Lawrence Berkeley National Laboratory, Technical Report LBNL-1014E,*, 2008.

[4] T. D. Arber, K. Bennett, C. S. Brady, M. G. Ramsaym, N. J. Sircombe, P. Gillies, R. G. Evans, H. Schmitz, A. R. Benn, and C. P. Ridgers. Contemporary Particle-in-Cell Approach to Laser-Plasma Modelling. *Plasma Physics and Controlled Fusion*, 2015.

[5] N. Arora, A. Shringarpure, and R. W. Vuduc. Direct N-body Kernels for Multicore Platforms. In *Proceedings of the International Conference on Parallel Processing*, ICPP '09, pages 379–387, Vienna, Austria, September 2009. IEEE Computer Society.

[6] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, et al. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[7] D. W. Barnette, R. F. Barrett, S. D. Hammond, J. Jayaraj, and J. H.

Laros III. Using miniapplications in a Mantevo framework for optimizing Sandia's SPARC CFD code on multi-core, many-core, and GPU-accelerated compute platforms. In *51st AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition*, page 1126, 2012.

[8] R. F. Bird, S. A. Wright, D. A. Beckingsale, and S. A. Jarvis. Performance Modelling of Magnetohydrodynamics Codes. In *Lecture Notes in Computer Science (LNCS), 7587:197-209*, July 2013.

[9] R. F. Bird, S. J. Pennycook, S. A. Wright, and S. A. Jarvis. Towards a Portable and Future-proof Particle-in-Cell Plasma Physics Code. *International Workshop on OpenCL*, 2014.

[10] R. F. Bird, P. Gillies, M. R. Bareford, and S. A. Jarvis. Mini-app Driven Optimisation of Inertial Confinement Fusion Codes. *IEEE Cluster Workshop on Representative Applications (WRAp)* , September 2015.

[11] C. K. Birdsall and A. B. Langdon. *Plasma physics via computer simulation*. CRC Press, 2014.

[12] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. John Wiley & Sons, 2006.

[13] R. Bordawekar, U. Bondhugula, and R. Rao. Believe it or Not! Multi-core CPUs Can Match GPU Performance for FLOP-intensive Application! Technical Report RC24982, IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY, 2010.

[14] R. Bordawekar, U. Bondhugula, and R. Rao. Can CPUs Match GPUs on Performance with Productivity?: Experiences with Optimizing a FLOP-intensive Application on CPUs and GPU. Technical Report RC25033, IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY, 2010.

[15] K. J. Bowers, B. Albright, L. Yin, B. Bergen, and T. Kwan. Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation. *Physics of Plasmas*, 15(5):055703, 2008.

[16] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Supercomputing, ACM/IEEE 2000 Conference*, pages 42–42. IEEE, 2000.

[17] H. Burau, R. Widera, W. Hönig, G. Juckeland, A. Debus, T. Kluge, U. Schramm, T. Cowan, R. Sauerbrey, and M. Bussmann. PIConGPU: A Fully Relativistic Particle-in-Cell Code for a GPU Cluster. *IEEE Transactions on Plasma Science*, 38(10):2831–2839, 2010. ISSN 0093-3813.

[18] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley. 24/7 characterization of petascale I/O workloads. In *IEEE International Conference on Cluster Computing and Workshops*, pages 1–10. IEEE, 2009.

[19] E. Carrier, A. Reisner, K. Czuprynski, R. Pavel, P. Grosset, and R. Bird. Evaluating Distributed Runtimes in the Context of Adaptive Mesh Refinement. *LANL Student Symposium, Los Alamos National Laboratory*, August 2014.

[20] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IEEE International Symposium on Workload Characterization*, pages 44–54. IEEE, 2009.

[21] G. Chen, L. Chacón, and D. C. Barnes. An efficient mixed-precision, hybrid CPU-GPU implementation of a nonlinearly implicit one-dimensional particle-in-cell algorithm. *Journal of Computational Physics*, 231(16): 5374–5388, 2012.

[22] R. Courant, K. Friedrichs, and H. Lewy. On the Partial Difference Equa-

tions of Mathematical Physics. *IBM Journal of Research and Development*, 11(2):215–234, 1967.

[23] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: towards a realistic model of parallel computation. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '93, pages 1–12, New York, NY, USA, 1993. ACM.

[24] J. A. Davis, G. R. Mudalige, S. D. Hammond, J. Herdman, I. Miller, and S. A. Jarvis. Predictive Analysis of a Hydrodynamics Application on Large-Scale CMP Clusters. In *International Supercomputing Conference (ISC11)*, volume 26 of *Lecture Notes in Computer Science (R&D)*, pages 175–185. Springer, Hamburg, Germany, June 2011.

[25] J. Dongarra and M. A. Heroux. Toward a New Metric for Ranking High Performance Computing Systems. *Sandia Report, SAND2013-4744*, 312, 2013.

[26] J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK users' guide*, volume 8. Siam, 1979.

[27] J. Dongarra et al. The International Exascale Software Project Roadmap. *International Journal of High Performance Computing Applications*, 25 (1):3–60, 2011.

[28] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra. From CUDA to OpenCL: Towards a Performance-Portable Solution for Multi-platform GPU Programming. Technical Report UT-CS-10-656, University of Tennessee, Knoxville, TN, 2010.

[29] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376. IEEE, 2011.

[30] M. J. Flynn. Very High-Speed Computing Systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.

[31] M. Geimer, F. Wolf, B. J. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, 2010.

[32] M. B. Giles, G. R. Mudalige, Z. Sharif, G. Markall, and P. H. Kelly. Performance analysis of the OP2 framework on many-core architectures. *SIGMETRICS Performance Evaluation Review*, 38(4):9–15, Mar. 2011. ISSN 0163-5999.

[33] P. Gillies, N. J. Sircombe, R. F. Bird, S. J. Pennycook, and S. A. Jarvis. The challenges of porting the Particle-in-cell code EPOCH to new architectures. *JOWOG 34, Sandia National Laboratory*, August 2013.

[34] P. Gillies, N. J. Sircombe, and R. F. Bird. Particle in Cell Simulations at AWE. *JOWOG 34, Lawrence Livermore National Laboratory*, April 2014.

[35] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A Call Graph Execution Profiler. In *ACM Sigplan Notices*, volume 17, pages 120–126. ACM, 1982.

[36] Graph500.org. Brief Introduction — Graph 500 . `http://www.graph500.org/`, September 2015.

[37] S. Green. Particle Simulation using CUDA. *NVIDIA whitepaper*, 2010.

[38] M. Griebel and P. Zaspel. A multi-GPU accelerated solver for the three-dimensional two-phase incompressible Navier-Stokes equations. *Computer Science - Research and Development*, 25:65–73, 2010. ISSN 1865-2034.

[39] J. L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[40] S. D. Hammond. *Performance Modelling and Simulation of High Performance Computing Systems*. PhD thesis, University of Warwick, Gibbet Hill Road, 10 2011.

[41] S. D. Hammond, G. R. Mudalige, J. A. Smith, S. A. Jarvis, J. A. Herdman, and A. Vadgama. WARPP: A Toolkit for Simulating High Performance Parallel Scientific Codes. In *2nd International Conference on Simulation Tools and Techniques (SIMUTools09)*, March 2009.

[42] S. D. Hammond, G. R. Mudalige, J. A. Smith, J. A. Davis, S. A. Jarvis, J. Holt, I. Miller, J. A. Herdman, and A. Vadgama. To Upgrade or not to Upgrade? Catamount vs. Cray Linux Environment. In *IEEE International Symposium on Parallel Distributed Processing, Workshops and PhD Forum (IPDPSW)*, pages 1 –8, april 2010.

[43] J. A. Herdman, W. P. Gaudin, D. Turland, and S. D. Hammond. Benchmarking and Modelling of POWER-7, Westmere, BG/P, and GPUs: An Industry Case Study. *ACM SIGMETRICS Performance Evaluation Review*, 38(4), March 2011.

[44] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving Performance via Mini-applications. *Sandia National Laboratories, Techical Report SAND2009-5574*, 2009.

[45] T. Hoefler, T. Schneider, and A. Lumsdaine. LogGP in theory and practice–An in-depth analysis of modern interconnection networks and benchmarking methods for collective operations. *Simulation Modelling Practice and Theory*, 17(9):1511–1521, 2009.

[46] A. Hoisie, O. Lubeck, and H. Wasserman. Performance and scalability analysis of teraflop-scale parallel architectures using multidimensional wavefront applications. *International Journal of High Performance Computing Applications*, 14(4):330–346, 2000.

[47] M. Itzkowitz and Y. Maruyama. HPC Profiling with the Sun Studio Performance Tools. In *Tools for High Performance Computing 2009*, pages 67–93. Springer, 2010.

[48] C. L. Janssen, H. Adalsteinsson, and J. P. Kenny. Using simulation to design extremescale applications and architectures: programming model exploration. *SIGMETRICS Performance Evaluation Review*, 38(4):4–8, Mar. 2011. ISSN 0163-5999.

[49] H.-W. Jin, S. Sur, L. Chai, and D. K. Panda. Lightweight kernel-level primitives for high-performance MPI intra-node communication over multi-core systems. In *IEEE International Conference on Cluster Computing*, pages 446–451. IEEE, 2007.

[50] R. Joseph, G. Ravunnikutty, S. Ranka, E. D'Azevedo, and S. Klasky. Efficient GPU Implementation for Particle in Cell Algorithm. In *IEEE International Parallel Distributed Processing Symposium (IPDPS)*, pages 395–406, 2011.

[51] H. Kaiser, M. Brodowicz, and T. Sterling. Parallex an advanced parallel execution model for scaling-impaired applications. In *Parallel Processing Workshops, 2009. ICPPW'09. International Conference on*, pages 394–401. IEEE, 2009.

[52] L. V. Kale and S. Krishnan. *CHARM++: a portable concurrent object oriented system based on C++*, volume 28. ACM, 1993.

[53] I. Karlin, A. Bhatele, B. L. Chamberlain, J. Cohen, Z. Devito, M. Gokhale, R. Haque, R. Hornung, J. Keasler, D. Laney, et al. LULESH Programming Model and Performance Ports Overview. *Lawrence Livermore National Laboratory (LLNL), Livermore, CA, Tech. Rep*, 2012.

[54] I. Karlin, J. McGraw, J. Keasler, and B. Still. Tuning the LULESH Mini-app for Current and Future Hardware. In *Nuclear Explosive Code Development Conference Proceedings (NECDC12)*, 2012.

[55] R. Kazman, M. Klein, and P. Clements. ATAM: Method for architecture evaluation. Technical report, DTIC Document, 2000.

[56] D. Kerbyson, A. Hoisie, and H. Wasserman. Modelling the performance of large-scale systems. *IEE Proceedings – Software*, 150(4):214, 2003. ISSN 14625970.

[57] Khronos OpenCL Working Group. OpenCL 1.2 Specification. `http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf`, 2015.

[58] V. Kindratenko and P. Trancoso. Trends in high-performance computing. *Computing in Science & Engineering*, 13(3):92–95, 2011.

[59] M. Klemm, A. Duran, X. Tian, H. Saito, D. Caballero, and X. Martorell. Extending OpenMP* with Vector Constructs for Modern Multicore SIMD Architectures. In *Proceedings of the International Workshop on OpenMP*, IWOMP '12, pages 59–72, Rome, Italy, 2012. Springer-Verlag.

[60] K. Knobe. Ease of use with concurrent collections (CnC). *Hot Topics in Parallelism*, 2009.

[61] K. Knobe and C. D. Offner. Tstreams: A model of parallel computation (preliminary report). Technical report, Technical Report HPL-2004-78, HP Labs, 2004.

[62] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa, and H. Kobayashi. Evaluating Performance and Portability of OpenCL Programs. In *Proceedings of the International Workshop on Automatic Performance Tuning*, iWAPT '11, Berkeley, CA, June 2010. Springer.

[63] J. Larus. Spending Moore's dividend. *Communications of the ACM*, 52 (5):62–69, 2009.

[64] P.-F. Lavallée, G. C. de Verdière, P. Wautelet, D. Lecas, and J.-M. Dupays. Porting and optimizing HYDRO to new platforms and programming paradigms–lessons learnt, 2012.

[65] S. H. Lavington. *A history of Manchester computers*. NCC Publications, 1975.

[66] Lawrenece Livermore National Laboratory. Hydrodynamics Challenge Problem. *Lawrence Livermore National Laboratory, Technical Report LLNL-TR-490254*, 2015.

[67] V. W. Lee et al. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture*, pages 451–460, 2010.

[68] Z. Lin, T. S. Hahm, W. Lee, W. M. Tang, and R. B. White. Turbulent Transport Reduction by Zonal Flows: Massively Parallel Simulations. *Science*, 281(5384):1835–1837, 1998.

[69] H. A. Lorentz. *Electromagnetic phenomena in a system moving with any velocity smaller than that of light*. Springer, 1937.

[70] B. Marder. A method for incorporating Gauss' law into electromagnetic PIC codes. *Journal of Computational Physics*, 68(1):48–55, 1987.

[71] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Technical Committee on Computer Architecture Newsletter*, page 19?25, 1995.

[72] S. McCartney. *ENIAC: The triumphs and tragedies of the world's first computer*. Walker & Company, 1999.

[73] P. Micikevicius. 3D Finite Difference Computation on GPUs using CUDA. In *Proceedings of the 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 79–84, New York, NY, USA, 2009. ACM.

[74] G. Moore. Cramming more components onto integrated circuits (from 1965). *Readings in computer architecture*, page 56, 2000.

[75] G. R. Mudalige, M. B. Giles, C. Bertolli, and P. H. Kelly. Predictive modeling and analysis of OP2 on distributed memory GPU clusters. In *Proceedings of the second international workshop on Performance modeling, benchmarking and simulation of high performance computing systems*, PMBS '11, pages 3–4, New York, NY, USA, 2011. ACM.

[76] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[77] NERSC. NERSC Benchmarking and Workload Characterisation). `http://www.nersc.gov/research-and-development/benchmarking-and-workload-characterization/`, 2015.

[78] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[79] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–13, New Orleans, LA, November 2010. IEEE Computer Society.

[80] A. Nichols. Users manual for ALE3D: An arbitrary Lagrange/Eulerian 3D code system. *Lawrence Livermore National Laboratory*, 2007.

[81] B. Nichols, D. Buttlar, and J. Farrell. *Pthreads programming: A POSIX standard for better multiprocessing*. " O'Reilly Media, Inc.", 1996.

[82] OpenACC Standards Committee. OpenACC 1.0 Specification. `http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf`, November 2011.

[83] OpenMP Standards Committee. OpenMP 4.0 Specification. `http://www.openmp.org/mp-documents/OpenMP4.0RC1_final.pdf`, November 2012.

[84] OpenMP Standards Committee. Technical Report on Directives for Attached Accelerators. Technical Report TR1, OpenMP Architecture Review Board, Champaign, IL, November 2012.

[85] B. Pang, U. li Pen, and M. Perrone. Magnetohydrodynamics on Heterogeneous architectures: a performance comparison. *CoRR*, abs/1004.1680, 2010.

[86] R. Pavel, R. Bird, P. Grosset, K. Czuprynski, A. Reisner, E. Carrier, C. Junghans, B. Bergen, and A. McPherson. Adaptive Mesh Refinement under the Concurrent Collections Programming Model. *In: The Sixth Annual Concurrent Collections Workshop*, 2014.

[87] J. Payne, D. Knoll, A. McPherson, W. Taitano, L. Chacon, G. Chen, and S. Pakin. Design and development of a multi-architecture, fully implicit, charge and energy conserving particle-in-cell framework. *Bulletin of the American Physical Society*, 58, 2013.

[88] S. J. Pennycook and S. A. Jarvis. Developing Performance-Portable Molecular Dynamics Kernels in OpenCL. In *Proceedings of the International Workshop on Performance Modeling, Benchmark and Simulation of HPC Systems*, PMBS '12, Salt Lake City, UT, November 2012.

[89] S. J. Pennycook, S. D. Hammond, S. A. Wright, J. A. Herdman, I. Miller, and S. A. Jarvis. An Investigation of the Performance Portability of OpenCL. *Journal of Parallel and Distributed Computing*, (to appear), 2012.

[90] S. J. Pennycook, S. D. Hammond, G. R. Mudalige, S. A. Wright, and S. A. Jarvis. On the Acceleration of Wavefront Applications using Distributed Many-Core Architectures . *The Computer Journal*, 55(2):138–153, August 2011.

[91] S. J. Pennycook, C. J. Hughes, M. Smelyanskiy, and S. A. Jarvis. Exploring SIMD for Molecular Dynamics, Using Intel® Xeon® Processors

and Intel® Xeon Phi Coprocessors. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 1085–1097. IEEE, 2013.

[92] O. Perks and D. A. Beckingsale. UK Mini-App Consortium (UK-MAC). `http://uk-mac.github.io`, 2015.

[93] O. Perks, S. D. Hammond, S. J. Pennycook, and S. A. Jarvis. WMTrace: a lightweight memory allocation tracker and analysis framework. In *UK Performance Engineering Workshop (UKPEW11)*, 2011.

[94] O. Perks, R. F. Bird, D. A. Beckingsale, and S. A. Jarvis. Exploiting spatiotemporal locality for fast call stack traversal. In *26th International Conference on Supercomputing*, 2012.

[95] R. Rabenseifner, G. Hager, and G. Jost. Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 427–436. IEEE, 2009.

[96] R. Reussner, P. Sanders, L. Prechelt, and M. Müller. SKaMPI: A detailed, accurate MPI benchmark. In *Recent advances in parallel virtual machine and message passing interface*, pages 52–59. Springer, 1998.

[97] A. D. Robison and R. E. Johnson. Three layer cake for shared-memory programming. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, page 5. ACM, 2010.

[98] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, and B. Jacob. The structural simulation toolkit. *SIGMETRICS Perform. Eval. Rev.*, 38 (4):37–42, Mar. 2011. ISSN 0163-5999.

[99] F. Rossi, P. Londrillo, A. Sgattoni, S. Sinigardi, and G. Turchetti. Robust Algorithms for Current Deposition and Dynamic Load-balancing in a GPU

Particle-in-Cell Code. *ADVANCED ACCELERATOR CONCEPTS: 15th Advanced Accelerator Concepts Workshop*, 1507(1):184–192, 2012.

[100] H. Ruhl. Classical Particle Simulations with the PSC Code, 2005.

[101] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 73–82, New York, NY, USA, 2008. ACM.

[102] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82. ACM, 2008.

[103] Y. Sentoku and A. J. Kemp. Numerical methods for particle simulations at extreme densities and temperatures: Weighted particles, relativistic collisions and reduced currents. *Journal of Computational Physics*, 227 (14):6846–6861, 2008.

[104] M. Smelyanskiy, D. Holmes, J. Chhugani, A. Larson, D. Carmean, D. Hanson, P. Dubey, K. Augustine, D. Kim, A. Kyker, V. Lee, A. Nguyen, L. Seiler, and R. Robb. Mapping High-Fidelity Volume Rendering for Medical Imaging to CPU, GPU and Many-Core Architectures. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1563–1570, November 2009. ISSN 1077-2626.

[105] G. Stantchev, W. Dorland, and N. Gumerov. Fast parallel Particle-To-Grid interpolation for plasma PIC simulations on the GPU. *Journal of Parallel and Distributed Computing*, 68(10):1339 – 1349, 2008. ISSN 0743-7315. General-Purpose Processing using Graphics Processing Units.

[106] E. Tejedor and R. M. Badia. Comp Superscalar: Bringing grid superscalar and GCM together. In *Cluster Computing and the Grid, 2008. CCGRID'08. 8th IEEE International Symposium on*, pages 185–193. IEEE, 2008.

[107] R. Teyssier. Cosmological Hydrodynamics with Adaptive Mesh Refinement: a new high resolution code called RAMSES. *Astronomy & Astrophysics*, 385(1):337–364, 2002.

[108] J. E. Thornton. The CDC 6600 project. *Annals of the History of Computing*, 2(4):338–348, 1980.

[109] TOP500.org. Home — TOP500 Supercomputer Sites. `http://www.top500.org/`, September 2015.

[110] TOP500.org. June 2015 — TOP500 Supercomputer Sites. `http://www.top500.org/lists/2015/06/`, June 2015.

[111] J. Uniejewski. SPEC Benchmark Suite: Designed for todays advanced systems. *SPEC Newsletter*, 1(1):1, 1989.

[112] B. Van Leer. Towards the ultimate conservative difference scheme. V. A second-order sequel to Godunov's method. *Journal of computational Physics*, 32(1):101–136, 1979.

[113] R. Vuduc, A. Chandramowlishwaran, J. Choi, M. E. Guney, and A. Shringarpure. On the Limits of GPU Acceleration. In *Proceedings of the USENIX Workshop on Hot Topics in Parallelism*, 2010.

[114] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick. Optimization of a Lattice Boltzmann Computation on State-of-the-Art Multicore Platforms. *Journal of Parallel and Distributed Computing*, 69(9):762–777, September 2009. ISSN 0743-7315.

[115] S. A. Wright, S. D. Hammond, S. J. Pennycook, R. F. Bird, J. Herdman, I. Miller, A. Vadgama, A. Bhalerao, and S. A. Jarvis. Parallel File System

Analysis Through Application I/O Tracing. *The Computer Journal*, 56 (2):141–155, 2013.

[116] K. S. Yee. Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media. *IEEE Trans. Antennas Propag*, 14(3):302–307, 1966.

Appendices

APPENDIX A
Performance-Portable Plasma Physics Simulations

Table A.1: Overhead introduced by atomic operations.

Hardware	With Atomics	Without Atomics
X5550	11.50	5.17
C1060	49.12	3.44
C2050	31.15	1.95
K20	3.04	1.41

Table A.2: Impact of st orting schemes on the Field Calculation kernel runtimes.

Hardware	Default	Sorted	Random
X5550	1.00	1.01	1.01
C1060	1.00	0.55	2.70
C2050	1.00	0.69	2.04
K20	1.00	0.80	2.07

Table A.3: Impact of sorting schemes on the Current Accumulation kernel runtimes.

Hardware	Default	Sorted	Random
X5550	1.00	0.96	0.95
C1060	1.00	4.51	0.13
C2050	1.00	3.40	0.45
K20	1.00	2.77	0.70

Table A.4: Best kernel performance across platforms.

Hardware	Current Accumulation	Field Calculation	Particle Move	Overhead
X5550 (F90)	5.39	1.48	0.04	0.00
X5550 (OpenCL)	6.44	2.64	0.13	0.78
E5-2670 (F90)	1.22	0.31	0.01	0.00
C1060	49.12	2.58	0.04	0.93
C2050	30.90	0.35	0.02	0.82
K20	3.04	0.23	0.01	0.78

APPENDIX B
Optimisation of Inertial Confinement Fusion Simulations

Table B.1: Normalised Data for Original EPOCH timestep scaling.

Step Number	Original	Modified
1	0.43	1.00
2	0.44	1.01
3	0.44	1.02
4	0.44	1.02
5	0.45	1.03
6	0.45	1.04
7	0.45	1.05
8	0.46	1.05
9	0.46	1.06
10	0.46	1.07
11	0.46	1.07
12	0.47	1.08
13	0.47	1.09
14	0.47	1.10
15	0.48	1.10
16	0.48	1.11
17	0.48	1.12
18	0.49	1.13
19	0.49	1.13
20	0.49	1.14
21	0.50	1.15
22	0.50	1.16
23	0.50	1.16
24	0.51	1.17
25	0.51	1.18
26	0.51	1.18
27	0.51	1.19
28	0.52	1.20
29	0.52	1.20
30	0.52	1.21
31	0.53	1.21
32	0.53	1.22
33	0.53	1.22
34	0.53	1.23
35	0.53	1.23
36	0.54	1.24
37	0.54	1.24
38	0.54	1.25
39	0.54	1.25
40	0.54	1.26

Continued on next page

Table B.1 Continued from previous page

Step Number	Original	Modified
41	0.54	1.26
42	0.55	1.26
43	0.55	1.27
44	0.55	1.27
45	0.55	1.27
46	0.55	1.28
47	0.55	1.28
48	0.55	1.28
49	0.55	1.28
50	0.55	1.28
51	0.56	1.28
52	0.56	1.28
53	0.56	1.29
54	0.56	1.29
55	0.56	1.29
56	0.56	1.29
57	0.56	1.29
58	0.56	1.29
59	0.56	1.29
60	0.56	1.29
61	0.56	1.29
62	0.56	1.29
63	0.56	1.29
64	0.56	1.29
65	0.56	1.29
66	0.56	1.29
67	0.56	1.29
68	0.56	1.29
69	0.56	1.29
70	0.56	1.29
71	0.56	1.29
72	0.56	1.29
73	0.56	1.30
74	0.56	1.30
75	0.56	1.30
76	0.56	1.30
77	0.56	1.30
78	0.56	1.30
79	0.56	1.30
80	0.56	1.30
81	0.56	1.30
82	0.56	1.30
83	0.56	1.30
84	0.56	1.30
85	0.56	1.30

Continued on next page

Table B.1 Continued from previous page

Step Number	Original	Modified
86	0.56	1.30
87	0.56	1.30
88	0.56	1.30
89	0.56	1.30
90	0.56	1.30
91	0.56	1.30
92	0.56	1.30
93	0.56	1.30
94	0.56	1.30
95	0.56	1.30
96	0.56	1.30
97	0.56	1.30
98	0.56	1.30
99	0.56	1.31
100	0.56	1.31
		Concluded

Table B.2: Cache Miss Counts for Array based EPOCH.

Step Number	L1 Misses	L2 Misses	L3 Misses
1	860,382.94	128,200.51	27,511.68
2	1,221,048.81	212,382.32	31,606.29
3	1,622,297.94	298,308.64	35,443.76
4	2,029,055.97	382,744.04	39,726.72
5	2,453,458.38	459,913.71	43,739.18
6	2,894,230.53	541,976.00	47,550.28
7	3,346,198.11	629,991.49	51,539.78
8	3,807,903.03	723,860.08	55,570.63
9	4,274,000.43	814,674.24	59,693.61
10	4,744,024.96	901,319.33	63,384.04
11	5,215,178.68	986,404.13	67,175.74
12	5,690,036.19	1,072,973.78	70,763.82
13	6,155,613.79	1,155,886.26	74,337.85
14	6,618,025.50	1,236,303.29	77,830.54
15	7,072,443.07	1,314,792.07	81,146.99
16	7,519,462.71	1,389,593.25	84,192.75
17	7,954,146.19	1,463,156.10	87,170.83
18	8,377,473.78	1,533,627.08	90,334.60
19	8,786,777.47	1,605,395.06	92,993.14
20	9,180,582.22	1,678,754.49	95,749.63
21	9,558,100.57	1,747,141.56	98,211.78
22	9,920,543.58	1,811,177.18	100,701.50
23	10,268,734.64	1,869,916.43	102,824.92
24	10,595,265.44	1,928,320.36	104,855.64

Continued on next page

Table B.2 Continued from previous page

Step Number	L1 Misses	L2 Misses	L3 Misses
25	10,904,519.06	1,986,603.40	106,757.53
26	11,194,580.79	2,041,140.04	108,457.82
27	11,464,287.18	2,090,723.67	110,113.29
28	11,713,427.76	2,134,189.35	111,653.03
29	11,946,034.78	2,176,247.08	113,341.76
30	12,157,997.99	2,210,351.22	114,854.29
31	12,350,195.99	2,239,469.82	116,075.74
32	12,524,977.92	2,266,945.63	117,487.00
33	12,678,532.97	2,290,149.75	118,616.03
34	12,814,882.50	2,310,300.97	119,583.82
35	12,930,492.88	2,328,167.42	120,578.44
36	13,025,393.14	2,343,452.10	121,252.17
37	13,101,824.25	2,358,516.92	121,874.15
38	13,159,027.40	2,375,264.50	122,309.97
39	13,194,486.49	2,392,112.38	122,450.38
40	13,206,538.49	2,406,699.29	122,708.28
41	13,203,085.13	2,420,378.57	122,899.33
42	13,185,903.65	2,429,503.63	122,833.85
43	13,167,342.99	2,433,463.83	122,714.35
44	13,153,442.38	2,434,746.06	122,663.47
45	13,140,906.53	2,434,503.85	122,845.99
46	13,121,923.47	2,436,437.14	122,989.54
47	13,083,172.11	2,433,376.24	122,970.46
48	13,036,876.14	2,432,905.50	123,063.68
49	12,984,345.57	2,432,336.72	123,307.49
50	12,939,887.40	2,430,924.01	123,501.75
51	12,901,067.42	2,430,107.13	123,565.74
52	12,869,519.61	2,431,653.61	123,791.71
53	12,841,755.19	2,428,575.10	123,852.01
54	12,821,058.60	2,426,573.25	124,018.67
55	12,808,318.04	2,427,266.69	124,146.38
56	12,799,362.06	2,427,030.85	124,320.92
57	12,794,489.10	2,426,482.97	124,304.67
58	12,794,134.85	2,426,255.54	124,497.00
59	12,794,902.19	2,426,268.29	124,689.44
60	12,798,289.75	2,426,427.96	124,677.07
61	12,805,754.63	2,426,693.67	124,942.24
62	12,815,357.61	2,426,120.53	124,891.75
63	12,830,095.29	2,427,681.25	125,002.49
64	12,841,245.67	2,427,669.38	125,015.07
65	12,852,986.50	2,429,998.19	124,999.81
66	12,863,564.35	2,431,527.39	125,066.14
67	12,871,530.42	2,433,272.28	125,022.50
68	12,873,200.86	2,432,837.65	125,138.22
69	12,874,425.79	2,433,355.97	125,180.85

Continued on next page

Table B.2 Continued from previous page

Step Number	L1 Misses	L2 Misses	L3 Misses
70	12,879,924.51	2,433,771.18	125,039.64
71	12,877,938.85	2,432,849.68	125,062.19
72	12,877,079.29	2,432,507.31	124,979.78
73	12,875,861.39	2,432,772.92	125,168.58
74	12,876,740.31	2,433,241.56	125,269.06
75	12,879,718.54	2,433,919.83	125,163.17
76	12,878,328.57	2,433,850.74	125,161.06
77	12,880,060.92	2,433,791.07	125,102.44
78	12,878,982.50	2,433,220.15	125,088.29
79	12,881,037.93	2,432,754.83	125,099.53
80	12,880,444.13	2,433,580.83	125,132.63
81	12,880,642.11	2,433,968.26	125,096.01
82	12,882,070.58	2,434,520.17	125,136.43
83	12,883,738.76	2,433,486.58	125,078.53
84	12,883,183.81	2,434,344.15	125,037.43
85	12,884,213.13	2,433,630.35	125,208.53
86	12,885,357.29	2,433,660.89	125,080.69
87	12,882,276.18	2,433,482.47	125,165.82
88	12,882,968.31	2,434,250.07	125,046.11
89	12,883,876.94	2,434,229.39	125,111.42
90	12,884,523.96	2,433,967.93	125,115.10
91	12,883,665.92	2,433,399.43	125,015.10
92	12,883,039.25	2,433,717.21	125,148.78
93	12,885,175.50	2,434,161.79	125,147.10
94	12,883,721.04	2,435,793.39	125,218.67
95	12,885,469.47	2,434,418.60	125,182.03
96	12,888,422.42	2,436,029.83	125,166.83
97	12,885,839.03	2,435,490.94	125,206.71
98	12,887,770.10	2,435,669.00	125,197.25
99	12,886,738.46	2,435,722.46	125,023.51
100	12,888,053.88	2,435,924.40	125,207.44
			Concluded

Table B.3: Normalised Cache Miss Counts for Array based EPOCH.

Step Number	L1 Misses	L2 Misses	L3 Misses
1	1.00	1.00	1.00
2	1.42	1.66	1.15
3	1.89	2.33	1.29
4	2.36	2.99	1.44
5	2.85	3.59	1.59
6	3.36	4.23	1.73
7	3.89	4.91	1.87
8	4.43	5.65	2.02
			Continued on next page

Table B.3 Continued from previous page

Step Number	L1 Misses	L2 Misses	L3 Misses
9	4.97	6.35	2.17
10	5.51	7.03	2.30
11	6.06	7.69	2.44
12	6.61	8.37	2.57
13	7.15	9.02	2.70
14	7.69	9.64	2.83
15	8.22	10.26	2.95
16	8.74	10.84	3.06
17	9.24	11.41	3.17
18	9.74	11.96	3.28
19	10.21	12.52	3.38
20	10.67	13.09	3.48
21	11.11	13.63	3.57
22	11.53	14.13	3.66
23	11.94	14.59	3.74
24	12.31	15.04	3.81
25	12.67	15.50	3.88
26	13.01	15.92	3.94
27	13.32	16.31	4.00
28	13.61	16.65	4.06
29	13.88	16.98	4.12
30	14.13	17.24	4.17
31	14.35	17.47	4.22
32	14.56	17.68	4.27
33	14.74	17.86	4.31
34	14.89	18.02	4.35
35	15.03	18.16	4.38
36	15.14	18.28	4.41
37	15.23	18.40	4.43
38	15.29	18.53	4.45
39	15.34	18.66	4.45
40	15.35	18.77	4.46
41	15.35	18.88	4.47
42	15.33	18.95	4.46
43	15.30	18.98	4.46
44	15.29	18.99	4.46
45	15.27	18.99	4.47
46	15.25	19.00	4.47
47	15.21	18.98	4.47
48	15.15	18.98	4.47
49	15.09	18.97	4.48
50	15.04	18.96	4.49
51	14.99	18.96	4.49
52	14.96	18.97	4.50
53	14.93	18.94	4.50

Continued on next page

Table B.3 Continued from previous page

Step Number	L1 Misses	L2 Misses	L3 Misses
54	14.90	18.93	4.51
55	14.89	18.93	4.51
56	14.88	18.93	4.52
57	14.87	18.93	4.52
58	14.87	18.93	4.53
59	14.87	18.93	4.53
60	14.88	18.93	4.53
61	14.88	18.93	4.54
62	14.89	18.92	4.54
63	14.91	18.94	4.54
64	14.93	18.94	4.54
65	14.94	18.95	4.54
66	14.95	18.97	4.55
67	14.96	18.98	4.54
68	14.96	18.98	4.55
69	14.96	18.98	4.55
70	14.97	18.98	4.54
71	14.97	18.98	4.55
72	14.97	18.97	4.54
73	14.97	18.98	4.55
74	14.97	18.98	4.55
75	14.97	18.99	4.55
76	14.97	18.98	4.55
77	14.97	18.98	4.55
78	14.97	18.98	4.55
79	14.97	18.98	4.55
80	14.97	18.98	4.55
81	14.97	18.99	4.55
82	14.97	18.99	4.55
83	14.97	18.98	4.55
84	14.97	18.99	4.54
85	14.97	18.98	4.55
86	14.98	18.98	4.55
87	14.97	18.98	4.55
88	14.97	18.99	4.55
89	14.97	18.99	4.55
90	14.98	18.99	4.55
91	14.97	18.98	4.54
92	14.97	18.98	4.55
93	14.98	18.99	4.55
94	14.97	19.00	4.55
95	14.98	18.99	4.55
96	14.98	19.00	4.55
97	14.98	19.00	4.55
98	14.98	19.00	4.55

Continued on next page

Table B.3 Continued from previous page

Step Number	L1 Misses	L2 Misses	L3 Misses
99	14.98	19.00	4.54
100	14.98	19.00	4.55
			Concluded

Table B.4: Cache Miss Counts for Linked List based EPOCH.

Step Number	L1 Misses	L2 Misses	L3 Misses
1	1,267,619.25	334,976.25	52,615.83
2	1,750,249.67	604,849.50	76,937.96
3	2,278,793.23	870,392.00	99,689.67
4	2,804,645.50	1,131,481.08	123,628.13
5	3,373,565.21	1,392,383.44	147,594.90
6	3,957,447.63	1,652,120.52	172,109.90
7	4,552,467.00	1,909,627.29	198,264.15
8	5,164,108.75	2,163,981.56	224,994.56
9	5,786,041.23	2,417,785.52	251,913.13
10	6,412,099.52	2,666,276.17	279,831.71
11	7,044,306.73	2,909,083.00	308,280.06
12	7,679,550.06	3,141,273.40	336,976.10
13	8,316,702.67	3,368,034.56	366,655.38
14	8,948,577.15	3,588,037.52	396,534.17
15	9,573,765.71	3,805,519.44	426,525.60
16	10,193,166.35	4,015,404.88	456,821.25
17	10,802,569.15	4,219,432.08	487,192.15
18	11,406,958.06	4,418,704.23	517,650.88
19	11,994,040.27	4,609,541.17	548,392.94
20	12,559,855.02	4,790,150.83	578,825.85
21	13,115,931.75	4,966,957.48	609,528.13
22	13,652,304.67	5,143,441.92	639,848.81
23	14,168,155.83	5,314,159.27	669,944.02
24	14,665,697.58	5,478,152.00	699,189.08
25	15,134,576.69	5,630,317.52	727,977.60
26	15,583,997.79	5,775,091.46	756,412.85
27	16,004,401.60	5,909,849.50	784,261.15
28	16,397,820.23	6,029,893.54	811,521.88
29	16,761,875.73	6,131,912.27	837,983.42
30	17,095,508.02	6,228,241.58	863,504.98
31	17,394,024.35	6,319,318.56	888,487.75
32	17,667,942.04	6,412,630.81	912,743.48
33	17,914,615.88	6,501,523.65	935,997.13
34	18,133,323.69	6,589,190.17	958,172.69
35	18,317,017.56	6,671,185.40	979,352.42
36	18,465,873.85	6,756,703.27	999,788.52
37	18,577,379.79	6,839,136.27	1,019,114.27

Continued on next page

Table B.4 Continued from previous page

Step Number	L1 Misses	L2 Misses	L3 Misses
38	18,652,425.69	6,915,436.40	1,037,336.67
39	18,683,899.15	6,980,330.85	1,055,036.54
40	18,679,984.17	7,033,876.94	1,071,895.73
41	18,636,745.73	7,079,897.90	1,087,759.69
42	18,567,443.52	7,119,748.35	1,103,161.56
43	18,502,166.31	7,160,131.38	1,117,971.23
44	18,443,315.90	7,192,755.46	1,132,283.29
45	18,401,715.83	7,219,419.79	1,145,625.71
46	18,361,408.04	7,237,781.40	1,157,541.77
47	18,314,040.73	7,250,044.60	1,167,148.52
48	18,253,321.27	7,258,735.23	1,174,438.33
49	18,191,588.23	7,267,880.73	1,180,355.38
50	18,135,692.02	7,272,309.06	1,185,468.60
51	18,090,035.06	7,275,690.85	1,190,285.77
52	18,046,217.48	7,278,696.71	1,194,532.67
53	18,013,758.96	7,277,982.69	1,198,480.40
54	17,984,650.10	7,278,395.79	1,202,015.00
55	17,974,307.17	7,282,856.13	1,205,440.60
56	17,965,223.29	7,285,898.69	1,208,361.90
57	17,961,508.90	7,286,042.29	1,211,112.29
58	17,967,258.13	7,288,849.17	1,213,541.71
59	17,973,993.00	7,289,397.06	1,215,700.52
60	17,989,792.79	7,292,744.25	1,217,915.13
61	18,006,951.94	7,294,415.42	1,219,810.33
62	18,026,327.19	7,295,367.88	1,221,454.98
63	18,054,605.27	7,301,276.13	1,223,032.15
64	18,077,134.33	7,305,980.75	1,224,435.40
65	18,099,884.23	7,309,940.35	1,225,905.25
66	18,113,843.04	7,312,495.42	1,227,309.83
67	18,125,666.77	7,314,988.31	1,228,924.38
68	18,127,877.67	7,317,093.77	1,230,652.73
69	18,132,946.88	7,321,755.79	1,232,749.73
70	18,132,735.25	7,323,489.17	1,235,018.02
71	18,142,610.83	7,329,166.04	1,237,645.13
72	18,145,797.15	7,330,608.79	1,240,428.69
73	18,148,000.69	7,334,864.79	1,243,379.71
74	18,156,556.77	7,339,337.00	1,246,398.67
75	18,158,393.25	7,343,754.96	1,249,379.35
76	18,164,600.38	7,347,334.92	1,252,348.73
77	18,169,893.92	7,350,248.06	1,255,192.42
78	18,175,100.04	7,353,785.75	1,258,110.65
79	18,181,845.67	7,357,708.38	1,260,984.54
80	18,187,996.44	7,361,043.71	1,263,858.08
81	18,189,868.27	7,363,845.83	1,266,578.19
82	18,193,891.75	7,366,221.73	1,269,253.25

Continued on next page

Table B.4 Continued from previous page

Step Number	L1 Misses	L2 Misses	L3 Misses
83	18,196,724.31	7,366,181.15	1,271,849.60
84	18,204,512.02	7,369,955.27	1,274,437.50
85	18,207,699.58	7,371,707.15	1,276,960.38
86	18,210,973.63	7,372,045.63	1,279,422.23
87	18,214,281.00	7,375,238.58	1,281,727.27
88	18,218,792.67	7,376,719.35	1,284,048.65
89	18,222,211.02	7,378,389.63	1,286,081.42
90	18,219,161.58	7,377,546.71	1,287,852.94
91	18,223,554.63	7,378,882.29	1,289,267.33
92	18,229,897.31	7,383,560.29	1,290,408.71
93	18,228,477.44	7,382,873.71	1,291,186.73
94	18,228,168.27	7,383,888.29	1,291,613.83
95	18,229,829.79	7,384,271.98	1,292,063.69
96	18,232,722.54	7,386,239.00	1,292,248.35
97	18,235,917.25	7,385,578.56	1,292,378.40
98	18,237,527.00	7,387,074.25	1,292,444.75
99	18,239,020.85	7,386,564.81	1,292,466.56
100	18,236,300.21	7,386,992.96	1,292,494.92

Concluded

Table B.5: Normalised Cache Miss Counts for Linked List based EPOCH.

Step Number	L1 Misses	L2 Misses	L3 Misses
1	1.00	1.00	1.00
2	1.38	1.81	1.46
3	1.80	2.60	1.89
4	2.21	3.38	2.35
5	2.66	4.16	2.81
6	3.12	4.93	3.27
7	3.59	5.70	3.77
8	4.07	6.46	4.28
9	4.56	7.22	4.79
10	5.06	7.96	5.32
11	5.56	8.68	5.86
12	6.06	9.38	6.40
13	6.56	10.05	6.97
14	7.06	10.71	7.54
15	7.55	11.36	8.11
16	8.04	11.99	8.68
17	8.52	12.60	9.26
18	9.00	13.19	9.84
19	9.46	13.76	10.42
20	9.91	14.30	11.00
21	10.35	14.83	11.58

Continued on next page

Table B.5 Continued from previous page

Step Number	L1 Misses	L2 Misses	L3 Misses
22	10.77	15.35	12.16
23	11.18	15.86	12.73
24	11.57	16.35	13.29
25	11.94	16.81	13.84
26	12.29	17.24	14.38
27	12.63	17.64	14.91
28	12.94	18.00	15.42
29	13.22	18.31	15.93
30	13.49	18.59	16.41
31	13.72	18.86	16.89
32	13.94	19.14	17.35
33	14.13	19.41	17.79
34	14.31	19.67	18.21
35	14.45	19.92	18.61
36	14.57	20.17	19.00
37	14.66	20.42	19.37
38	14.71	20.64	19.72
39	14.74	20.84	20.05
40	14.74	21.00	20.37
41	14.70	21.14	20.67
42	14.65	21.25	20.97
43	14.60	21.38	21.25
44	14.55	21.47	21.52
45	14.52	21.55	21.77
46	14.48	21.61	22.00
47	14.45	21.64	22.18
48	14.40	21.67	22.32
49	14.35	21.70	22.43
50	14.31	21.71	22.53
51	14.27	21.72	22.62
52	14.24	21.73	22.70
53	14.21	21.73	22.78
54	14.19	21.73	22.85
55	14.18	21.74	22.91
56	14.17	21.75	22.97
57	14.17	21.75	23.02
58	14.17	21.76	23.06
59	14.18	21.76	23.11
60	14.19	21.77	23.15
61	14.21	21.78	23.18
62	14.22	21.78	23.21
63	14.24	21.80	23.24
64	14.26	21.81	23.27
65	14.28	21.82	23.30
66	14.29	21.83	23.33

Continued on next page

Table B.5 Continued from previous page

Step Number	L1 Misses	L2 Misses	L3 Misses
67	14.30	21.84	23.36
68	14.30	21.84	23.39
69	14.30	21.86	23.43
70	14.30	21.86	23.47
71	14.31	21.88	23.52
72	14.31	21.88	23.58
73	14.32	21.90	23.63
74	14.32	21.91	23.69
75	14.32	21.92	23.75
76	14.33	21.93	23.80
77	14.33	21.94	23.86
78	14.34	21.95	23.91
79	14.34	21.96	23.97
80	14.35	21.97	24.02
81	14.35	21.98	24.07
82	14.35	21.99	24.12
83	14.36	21.99	24.17
84	14.36	22.00	24.22
85	14.36	22.01	24.27
86	14.37	22.01	24.32
87	14.37	22.02	24.36
88	14.37	22.02	24.40
89	14.38	22.03	24.44
90	14.37	22.02	24.48
91	14.38	22.03	24.50
92	14.38	22.04	24.53
93	14.38	22.04	24.54
94	14.38	22.04	24.55
95	14.38	22.04	24.56
96	14.38	22.05	24.56
97	14.39	22.05	24.56
98	14.39	22.05	24.56
99	14.39	22.05	24.56
100	14.39	22.05	24.56

Concluded

Table B.6: Data for Original and Modified EPOCH timestep scaling.

Step Number	Original	Modified
1	0.43	0.43
2	0.44	0.43
3	0.44	0.43
4	0.44	0.43
5	0.45	0.43

Continued on next page

Table B.6 Continued from previous page

Step Number	Original	Modified
6	0.45	0.43
7	0.45	0.43
8	0.46	0.43
9	0.46	0.44
10	0.46	0.44
11	0.46	0.44
12	0.47	0.44
13	0.47	0.44
14	0.47	0.44
15	0.48	0.44
16	0.48	0.44
17	0.48	0.44
18	0.49	0.44
19	0.49	0.44
20	0.49	0.45
21	0.50	0.45
22	0.50	0.45
23	0.50	0.45
24	0.51	0.45
25	0.51	0.45
26	0.51	0.45
27	0.51	0.45
28	0.52	0.45
29	0.52	0.45
30	0.52	0.45
31	0.53	0.45
32	0.53	0.45
33	0.53	0.45
34	0.53	0.45
35	0.53	0.45
36	0.54	0.45
37	0.54	0.45
38	0.54	0.45
39	0.54	0.45
40	0.54	0.45
41	0.54	0.45
42	0.55	0.45
43	0.55	0.45
44	0.55	0.45
45	0.55	0.45
46	0.55	0.45
47	0.55	0.45
48	0.55	0.45
49	0.55	0.45
50	0.55	0.45

Continued on next page

Table B.6 Continued from previous page

Step Number	Original	Modified
51	0.56	0.45
52	0.56	0.45
53	0.56	0.45
54	0.56	0.45
55	0.56	0.45
56	0.56	0.45
57	0.56	0.45
58	0.56	0.45
59	0.56	0.45
60	0.56	0.45
61	0.56	0.45
62	0.56	0.45
63	0.56	0.45
64	0.56	0.45
65	0.56	0.45
66	0.56	0.45
67	0.56	0.45
68	0.56	0.45
69	0.56	0.45
70	0.56	0.45
71	0.56	0.45
72	0.56	0.45
73	0.56	0.45
74	0.56	0.45
75	0.56	0.45
76	0.56	0.45
77	0.56	0.45
78	0.56	0.45
79	0.56	0.45
80	0.56	0.45
81	0.56	0.45
82	0.56	0.45
83	0.56	0.45
84	0.56	0.45
85	0.56	0.45
86	0.56	0.45
87	0.56	0.45
88	0.56	0.45
89	0.56	0.45
90	0.56	0.45
91	0.56	0.45
92	0.56	0.45
93	0.56	0.45
94	0.56	0.45
95	0.56	0.45

Continued on next page

Table B.6 Continued from previous page

Step Number	Original	Modified
96	0.56	0.45
97	0.56	0.45
98	0.56	0.45
99	0.56	0.45
100	0.56	0.45
		Concluded

Table B.7: SIMD scaling of miniEPOCH AoS kernels for 256-bit AVX, 128-bit AVX and for the code operating without vectorisation.

Kernel	256-bit	128-bit	No Vec
Move	$2.27 \cdot 10^{-2}$	$2.35 \cdot 10^{-2}$	$2.37 \cdot 10^{-2}$
Stencil	$4.44 \cdot 10^{-2}$	$5.00 \cdot 10^{-2}$	$8.33 \cdot 10^{-2}$
Current	$6.42 \cdot 10^{-2}$	$8.19 \cdot 10^{-2}$	$1.14 \cdot 10^{-1}$

Table B.8: SIMD scaling of miniEPOCH SoA kernels for 256-bit AVX, 128-bit AVX and for the code operating without vectorisation.

Kernel	256-bit	128-bit	No Vec
Move	$7.66 \cdot 10^{-3}$	$7.69 \cdot 10^{-3}$	$7.69 \cdot 10^{-3}$
Stencil	$4.83 \cdot 10^{-2}$	$4.65 \cdot 10^{-2}$	$7.55 \cdot 10^{-2}$
Current	$4.69 \cdot 10^{-2}$	$6.54 \cdot 10^{-2}$	$9.84 \cdot 10^{-2}$

Table B.9: Normalised instruction counts per kernel for different SIMD widths for the SoA memory layout.

Kernel	256-bit	128-bit	No Vec
Move	0.25	0.50	1.00
Stencil	0.52	0.63	1.00
Current	0.37	0.59	1.00

Table B.10: Kernel runtime for different data layouts.

Kernel	**SoA**	**AoS**	**AoSoA**
Move	$7.66 \cdot 10^{-3}$	$2.27 \cdot 10^{-2}$	$1.73 \cdot 10^{-2}$
Stencil	$4.83 \cdot 10^{-2}$	$4.44 \cdot 10^{-2}$	$3.90 \cdot 10^{-2}$
Current	$4.69 \cdot 10^{-2}$	$6.42 \cdot 10^{-2}$	$6.56 \cdot 10^{-2}$

Table B.11: Overall runtime for the original and optimised versions of EPOCH for a 10,000 step run.

Code Variant	**Total Kernel Time**	**Sort Overhead**
Original	5332.19	0.00
Array	4502.09	0.00
Optimised	2164.41	1302.36

Table B.12: Particle Per Cell Scaling of EPOCH and miniEPOCH.

Particles Per Cell	**EPOCH**	**miniEPOCH**
32	0.22	0.25
64	0.43	0.40
128	0.87	0.70
256	1.74	1.31
512	3.47	2.46

Table B.13: Counts for the number of instructions executed during particle per cell scaling.

Particles Per Cell	**EPOCH**	**miniEPOCH** (vec)	**miniEPOCH** (novec)
32	1.00	1.00	1.00
64	2.00	1.54	1.95
128	4.00	2.57	3.84
256	8.00	4.65	7.65
512	16.00	8.80	15.26

APPENDIX C
Discussion and Conclusions

Table C.1: Kernel Performance of Intel Xeon Phi and Comparable CPU.

Kernel	**CPU**	**Intel Xeon Phi**
Move	$1.02 \cdot 10^{-3}$	$2.67 \cdot 10^{-3}$
Stencil	$2.47 \cdot 10^{-2}$	$2.70 \cdot 10^{-2}$
Current	$1.48 \cdot 10^{-2}$	$3.58 \cdot 10^{-2}$

Table C.2: Total Runtime of Intel Xeon Phi and Comparable CPU.

Code Variant	**Total Runtime**
EPOCH	$7.19 \cdot 10^{-2}$
miniEPOCH (Xeon Phi)	$6.54 \cdot 10^{-2}$
miniEPOCH (CPU)	$4.05 \cdot 10^{-2}$